Intro to Theory Computation Notes

New Beginnings, Summer 2018

David Lu

August 26, 2018

Contents

1 Theory of Computation

The three closely related and central topics in the theory of computation include complexity, computability, and mathematical models of computation or abstract machines. I've mentioned ideas about complexity and computability in class and in a few examples. The undergraduate theory of computation class at PSU (CS311) is primarily a class about abstract machines. The graduate theory of computation class (CS581) is concerned more with diving in to the details of computability and complexity. Both graduate and undergraduate algorithms courses (CS584 and CS350) naturally discuss complexity as well.

2 Alphabets

We define an alphabet as a finite, non-empty set of tokens, typically characters or short strings.

Examples:

- The letters, `a-z`
- The numbers, `0-9`
- The binary alphabet, {0, 1}
- Morse code alphabet, {·, -}
- NES controller inputs, $\{\leftarrow, \rightarrow, \uparrow, \downarrow, b, a, start, select\}$

Since it's relatively rare that many alphabets are of interest in a context, we often use greek letters Σ and Γ to refer to an alphabet under consideration.

3 Strings and Words

Strings or *words* over some alphabet refer to finite sequences of tokens from the alphabet.

3.1 Examples

- 10101
- $\uparrow\uparrow\downarrow\downarrow\leftarrow\rightarrow\leftarrow\rightarrow$ BA

We often refer to arbitrary strings as s or w, sometimes with a subscript to distinguish between them.

3.2 String length

The length of a string s, written $|s|$, is the number of alphabet tokens it contains. Strings of length 0 are allowed – we refer to them as the *empty string*, often abbreviated as ε.

3.3 An example string operation

The *concatenation* of strings s and t – that is, the sequence of symbols comprising s followed immediately by the sequence of symbols comprising t – is simply written st.
Concatenating a string s with itself, yields s^2. More generally, s concatenated with itself n times is written s^n. As an example, we could express the string $ababab$ as ab^3.

4 Languages

A language is a set of strings.

4.1 Examples

- $\{1001\}$
- The set of all strings composed of zero or more ab
- $\{w —w$ ends with a $1\}$
- The set of binary strings that contain 010 as a substring
- The language of any programming language – Part of your compiler's job involves checking to ensure that the programs you write consist of valid strings of the programming language

4.2 More examples

Recall that we can use the binary alphabet to encode all kinds of data, including numbers, characters, programs, and images. So less intuitive examples of languages include:

- The set of all valid bitmap images
- The set of all valid bitmap images that depict cats

5 Operations on Alphabets and Languages

Since languages and alphabets are fundamentally sets, all of the set operations that we discussed previously apply. Here are some additional operations we can perform:

5.1 Concatenation

We've briefly described the concatenation operation on strings. We can also concatenate languages. The concatenation of two languages A and B, written $A \circ B$, is defined as follows:

$$A \circ B = \{st | s \in A \text{ and } t \in B\}$$

In English, the language $A \circ B$ is the set of all strings that can be created by taking a string from A and concatenating it with a string from B.[1]

5.2 Kleene Star: *

We noted above that we can build strings from repeated substrings using the exponent notation. We can generalize this for languages: given a string s, we can generate a language $S = s^*$. Then S is the set of all strings that are zero or more concatenated repetitions of s.

An superscript star, '*', used to express the concept of "zero or more repetitions" is known as the Kleene star or Kleene closure after its creator, Stephen Kleene.

5.3 Kleene star to other concepts

We can generalize this idea to other related items. The Kleene closure over an alphabet, written Σ^* is the language containing every string that can be generated from zero or more tokens from Σ.

5.4 Alternate definition of a language

This gives us an alternate definition of a language, equivalent to the earlier definition: A language over an alphabet Σ is a subset of its Kleene closure, Σ^*.

We can also apply the Kleene closure to languages. For some language L, L^* is the language constructed by concatenating zero or more strings in L.[2]

[1] Note that if $\varepsilon \in A$, then $B \subseteq A \circ B$, and if $\varepsilon \in B$ then $A \subseteq A \circ B$.

[2] Note that because the Kleene closure represents *zero* or more concatenated elements, any Kleene closure includes the empty string ε, regardless of whether the start is applied to a string, alphabet, or language.

6 State Machines

I've been studying CS for a while, and I'm starting to burn out. So I've decided to get out of the game and start a new business venture in the wonderful world of tango dancing.
I know literally nothing about tango other than that I've heard the phrase "it takes two to tango." Therefore, I'd like to ensure that:

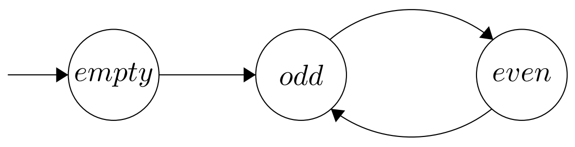
- I have a non-zero number of participants
- The number of participants is even

I have no idea how wildly successful my first event is going to be, so I'll be leaving registration open for an as-yet-undetermined period of time to see how it goes. Since I've sworn off computers forever, I'll be accepting registrations on paper. As they roll in, I'll pass them to you. At any point, I may ask you whether or not our registration conditions have been satisfied to see if we can proceed with the event.

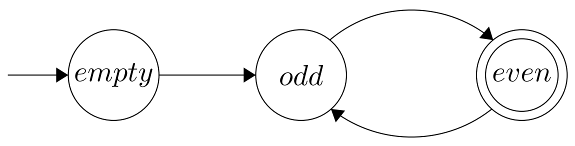
There are a couple of ways that you could approach this task. One way – which is the method we'd use if we were tracking these registrations in a database – would be to simply accept all registrations, count them, and see if the result is divisible by two.

We don't actually care about any properties of the count beyond the ones we've created though. So a simpler (but equally correct) method of accomplishing the same task would be to track *only* these properties. We can discard the value of the count and track only its non-equality with zero and its divisibility by two.

To do this, we might start by saying that our count is zero. When we see our first registration, we know that our count becomes odd. When we see our second, we know the count becomes even. With each additional registration, we flip back and forth between these two states. We can draw a *state diagram* for this process like so:



The arrow pointing into the "empty" state indicates that this is the *start state.* If we want one or more states to represent an acceptable state, we draw it in a double circle. Since only the even state matches all of our criteria, that's the only one that's acceptable. Like so:



This is an example of a *finite state machine*, a simple example of an abstract machine.

6.1 DFAs

A state machine is a model of computation. It is conceived as an abstract machine that can be in one of a finite number of states. We'll look at a specific type of state machine called a *deterministic finite automaton* or DFA for short. A DFA is in only one state at a time, its *current* state. It can change from one state to another through a triggering event or condition, called a *transition*. A particular DFA is defined by its states and the transitions for each state.

DFAs are used to implement predicates that takes a string as input and decides either to *accept* or *reject* it. Every properly constructed machine M accepts or *decides* a set of strings, which is called the language of that machine, denoted $L(M)$.

$L(M)$ is a function of M. In other words, the language of a machine is defined by that machine – not the other way around.

In the example above, we can think of this machine as deciding a language over a singleton alphabet of even, nonempty strings. We can describe the language as follows:

$$L(M) = \{s \mid |s| \text{ is even }\}$$

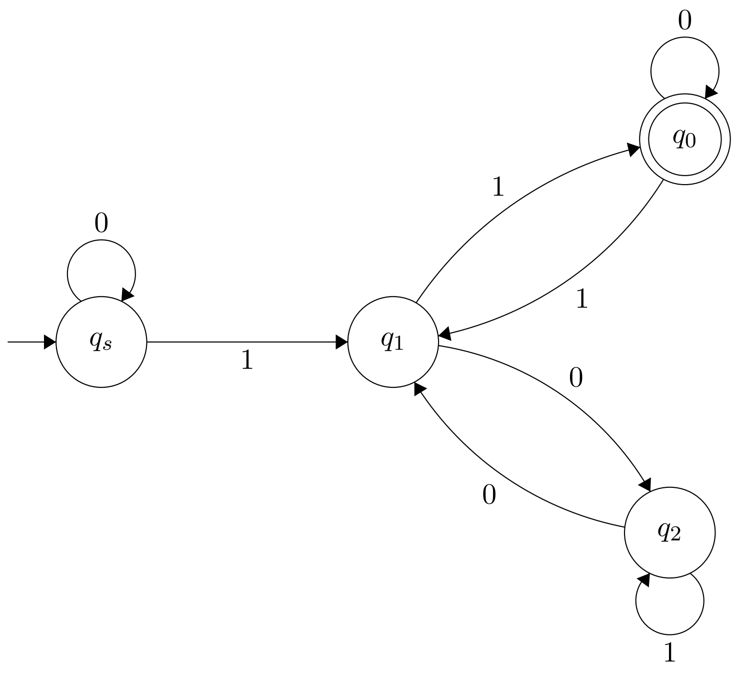
In other words, the language of this machine accepts strings such as 'rrrr', an 'r' for each registration. It won't accept the empty string, denoting zero registrants, and it won't accept strings such as 'rrr' denoting an odd number of registrants.
More generally, if the alphabet contains more than one token, we need to annotate each transition with the token that triggers it.

6.1.1 Examples: DFAs

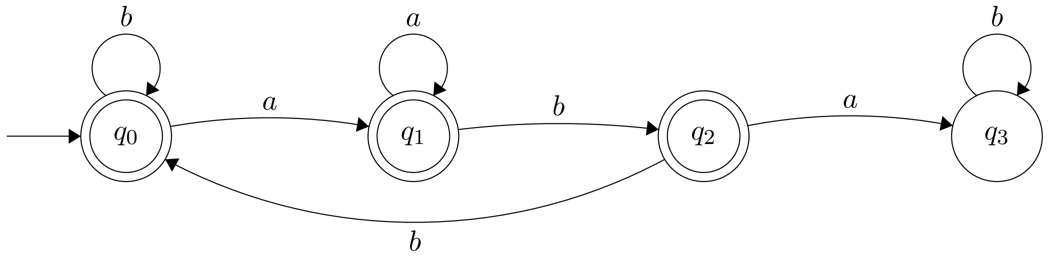
Strings that End in 111 Over a Binary Alphabet $L(M) = \{s \in (0 \mid 1)^* \mid s \text{ ends in } 111\}$ where $\Sigma = \{0, 1\}$

$$\begin{tikzcd}
{} \arrow[r] & q_0 \arrow[loop above, "0"] \arrow[r, bend left, "1"] & q_1 \arrow[l, bend left, "0"] \arrow[r, "1"] & q_2 \arrow[ll, bend right, "0"'] \arrow[r, "1"] & \boxed{q_3} \arrow[loop right, "1"] \arrow[lll, bend left=40, "0"]
\end{tikzcd}$$

Binary Multiples of Three $L(M) = \{s \in (0 \mid 1)^* \mid s \text{ is a binary multiple of } 3\}$ where $\Sigma = \{0, 1\}$
Drawing this DFA is much easier if we consider that every binary string represents some integer number, and that for any $x \in \mathbb{Z}, x \bmod 3 \in \{0, 1, 2\}$. Taking an existing binary number and adding a 1 to the end is equivalent to multiplying it by two and then adding 1; adding 0 to the end is equivalent to multiplying it by two.



Strings Over $\{a, b\}$ That Do Not Include Substring aba $L(M) = \{s \in (a \mid b)^* \mid s$ does not include the substring $aba\}$ where $\Sigma = \{a, b\}$



6.1.2 Determinism

A state machine is a *deterministic* finite automaton if it has:

- a deterministic set of transitions, i.e. a single transition for a given token of the alphabet for each state.
- a transition for *every* character from *every* state. In other words, we must define a transition for any possible input at every state.

6.1.3 Formalization

Drawing DFAs works well for small, simple languages like the examples above. What happens if a DFA that is unmanageably large? We can formalize the notion of a DFA as a collection of mathematical constructs for scalability.

A DFA is a 5-tuple, $(Q, \Sigma, \delta, q_0, F)$, where:

- Q is a set of states
- Σ is an alphabet
- $\delta : (Q \times \Sigma) \to Q$ is a transition function
- $q_0 \in Q$ is the start state
- $F \subseteq Q$ is a set of accepting states.

Example: Binary Strings Ending in 111 We need 4 states, each one indicating the number of 1s in a row we've seen so far. That can be 0, 1, 2, or 3. So $Q = \{q_0, q_1, q_2, q_3\}$.

We're dealing with binary strings, so $\Sigma = \{0, 1\}$.

The start state is q_0.

The accepting states $F = \{q_3\}$.

We can describe the transition function, δ, as a computation:

- From any state q_i, if next character in the input is a 0, transition to q_0
- From any state q_i, for $i \in [0, 2]$, if the next input is a 1, transition to q_{i+1}
- From q_3, if next input is a 1, stay at q_3

Or we might express the transition function as a table:

q	*input*	q_{next}
q_0	0	q_0
q_0	1	q_1
q_1	0	q_0
q_1	1	q_2
q_2	0	q_0
q_2	1	q_3
q_3	0	q_0
q_3	1	q_3

6.1.4 DFAs as a Model of Computation

DFAs provide a first model of computation. *Any* computable problem can be reduced to a problem of set membership. Since we can encode anything that can be encoded using a binary alphabet, we can reduce any computable problem to a language decision over a binary alphabet.

DFAs are not capable of computing *every* computable problem, however, but they can compute a subset thereof. Perhaps later you'll see more powerful abstract machines such as pushdown automata and Turing machines that can decide more languages than our DFAs.

Formally, a DFA $(Q, \Sigma, \delta, q_0, F)$ accepts s if Q contains a sequence of states $(r_0, r_1, ..., r_n)$ such that:

1. $r_0 = q_0$
2. $\delta(q_i, s_{i+1}) = r_{i+1}$ for $i \in [0, n)$
3. $r_n \in F$

6.2 State Machines for Other Things

DFAs recognize exactly the set of regular languages. This is useful for things such as pattern matching and lexical analysis. For instance, a DFA can decide whether or not user-input is a valid email address or not. For another example, consider that every compiler has a part called the *lexer* or *tokenizer*. The job of the lexer is to classify sequences of characters, e.g. to determine whether some sequence of characters is an operator, an identifier, or a keyword. Popular lexers, such as those generated by Flex and Lex, generate finite state machines internally.

State machines are also commonly used to sequence animations for characters or other animated game objects in video games.

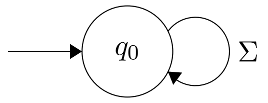
7 Regular Languages

Now that we have a basic picture of how DFAs work, we can introduce the first class of languages that interest us as computer scientists. We say that a language is *regular* if and only if there exists a DFA that recognizes it.

Proving that a language is regular is then as simple as constructing a DFA that recognizes it. It's important to note that failing to construct a DFA that recognizes a language in question does *not* prove non-regularity. To do that, we would need to *prove* that there cannot exist any DFA that recognizes it.

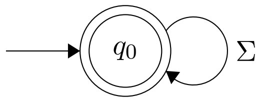
7.1 Two Easy Proofs

$\varnothing$ is a Regular Language The empty set is a regular language, since we can construct the following DFA:



As a shorthand, we've simply written Σ on the transition so that this DFA is alphabet-agnostic. It has a single non-accepting state and transitions to it on any given input from the alphabet.

Σ^* is a Regular Language By the same reasoning, we can prove that Σ^* is regular by constructing a very similar DFA:



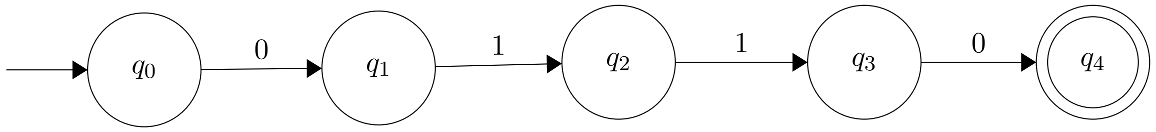
7.2 Proof by Construction

Thanks to our definition of a regular language, proving the regularity of a language is straightforward. Constructive proof allows us to generalize our reasoning to prove the regularity of whole sets of languages. To do this, we describe a method or algorithm that one could follow to create a DFA for any language in the set.

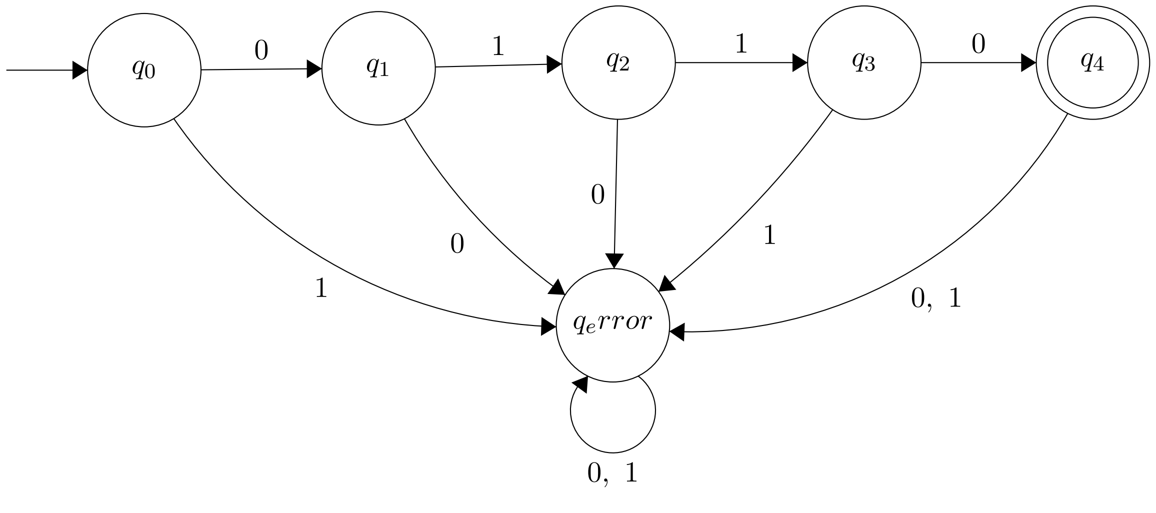
7.2.1 Example Proof: Any Singleton Language is Regular

Any language that contains only a single string is regular. We'll look at a specific example and then we'll generalize our reasoning.

Example: $L = \{0110\}$ Consider the language $L = \{0110\}$. It's a regular language, so a DFA exists that recognizes it. Constructing a DFA that recognizes L is pretty intuitive: we require a transition for each character with states between them as appropriate. We begin with something like this:



However, this is not a correctly specified DFA, since we haven't defined a transition for every character from every state. Since L contains just one string, any other transition than the ones we have results in an unsalvageable error. We can thus add a non-accepting error state to catch these cases.



Now every state has a transition for every possible input character; and we've shown L is regular by construction.

7.2.2 Generalizing the Proof

We can generalize on our method for constructing the DFA in the previous section.

Let $L = \{s\}$ be a language containing only one string, s. We can then construct a DFA that recognizes L using the following method.

Consider s as a sequence of characters, $(a_1, a_2, ..., a_n)$. Then:

- $Q = \{q_0\} \cup \{q_1, q_2, ..., q_n\} \cup \{q_{error}\}$
- $\Sigma = \{x \mid x \text{ appears in } s\}$
- $q_0 = q_0$
- $F = \{q_n\}$

We can define the transition function, δ, which takes a pair $< q, x >$ such that $q \in Q$ and $x \in \Sigma$ as follows:

- If $q = q_i$ for some $i \in [0, n)$, then
  –If $x = a_{i+1}$, transition to q_{i+1}
  –Else, transition to q_{error}
- If $q = q_n$, transition to q_{error}, regardless of the value of x
- If $q = q_{error}$, loop back to q_{error}, regardless of the value of x

Since we have described a general method for constructing DFAs that recognize L, L is a regular language.

8 Closure Properties

A set is closed under some operation if performing that operation on any element of the set yields another element of the set — that is to say, you can never escape the set by performing that operation on one of its elements. For instance:

- $\mathbb{N}$ is closed under addition and multiplication, but not subtraction or division
- $\mathbb{N}$ is closed under addition, multiplication, and subtraction, but not division
- $\mathbb{Q}$ and $\mathbb{R}$ are closed under addition, multiplication, subtraction, and division, because:
  - We can add and subtract fractions and ensure that the result is always a fraction by finding a common denominator

  - Multiplying fractions is as easy as taking the product of the numerators over the product of the denominators, which also yields a fraction
  - We can divide fractions by multiplying by a reciprocal

8.1 Exercise: Prove that Regular Languages are Closed under Complement

How can we use a constructive proof to prove that the regular languages are closed under the complement operation? This means that the complement of a regular language is always regular.