Theory of Computation Notes: Regular Languages and Finite Automata

Arjun Chandrasekhar

1 Alphabets, strings, and formal languages

Definition 1.1. An **alphabet** is a collection of symbols, e.g.

- $\{a, b\}$
- $\{0, 1\}$
- $\{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$

Definition 1.2. A **string** is a sequence of symbols from an alphabet, e.g.

- abbabababa
- 01010101010101
- 39074932749327

Let ϵ denote the **empty string**, i.e. the string with no symbols.

If Σ is an alphabet, we denote Σ^* to be the set of all possible strings that can be created from that alphabet. For example, if $\Sigma = \{a, b\}$ then $\Sigma^* = \{\epsilon, a, b, aa, ab, ba, bb, aaa, \dots\}$

Definition 1.3. Let Σ be an alphabet. A **formal language** is a collection of strings $L \subseteq \Sigma^*$, e.g.

- L = {w | w starts with 'a' and ends with 'b'} $\subseteq \{a, b\}^*$
- L = {w | w contains 001 as a substring} $\subseteq \{0, 1\}^*$
- L = {w | w is a multiple of 27} $\subseteq \{0, 1, \dots, 9\}^*$

2 Decision problems

Definition 2.1. A **function problem** is a problem in which we are given an input and produce an output, i.e.

- Given an integer x, output $x!$
- Given a graph G, output its chromatic number k
- Given a graph G and vertices u, v output the length of the shortest path from u to v in G

Definition 2.2. By contrast, a **decision problem** is a problem in which we are given an input and output either ACCEPT or REJECT, i.e.

- Given two integers x, y, decide if $y = x!$
- Given a graph G and an integer k, decide if k is the chromatic number of G
- Given a connected graph G, vertices u, v, and an integer k, decide if the shortest path from u to v has length $\leq k$

Essentially, a function problem is given an input and produces an output; a decision problem is given an input and its hypothesized output, and decides if the output is correct. If we can solve a function problem, can we solve its corresponding decision problem, and vice versa?

Let's take a look at the shortest path problem. First, let's suppose we had a black box to solve the function problem: given a graph G and vertices u, v, the black box outputs the shortest path length from u to v. Can we solve the corresponding decision problem: given a graph G, integers u, v, and an integer k, decide if the shortest path length from u to v is $\leq k$?

Yes! We simply feed G, u, v to our black box; the black box outputs a path length k'; we check if $k' \leq k$. Now let's check if the other direction holds. Suppose we have a magic black box that takes G, u, v, k as input and outputs ACCEPT if the shortest path length from u to v is $\leq k$, and REJECT otherwise. Can we determine the shortest path length from u to v?

Again, the answer is yes! Let $|V|$ be the total number of vertices in G. Then clearly the highest possible path length from u to v is $|V|$. So we iterate from $k = 1, 2, \ldots, |V|$. For each k, we feed G, u, v, k to our black box. If it outputs REJECT, we keep going. If it outputs ACCEPT for some k, then the shortest path length is $k - 1 < l \leq k$, meaning the shortest path length is k.

In general, it can be shown that every function problem can be converted to a corresponding decision problem. We will not show that here, but hopefully the shortest path length example is illustrative.

Why do we actually care about decision problems? The answer is, decision problems are the link between formal languages and algorithms. Every formal language has an associated decision problem. Let L be a formal language. The **decision problem associated with** L is the problem of taking a string w as input, and outputting ACCEPT if $w \in L$, and REJECT otherwise.

3 Computation

Definition 3.1. An **algorithm** is, intuitively, a mechanical procedure that takes an input and produces an output. An algorithm does not require thinking: it simply consists of step-by-step rules to follow mechanically until arriving at an output.

Definition 3.2. A **computer** is, intuitively, a machine that can carry out an algorithm. This class will be dedicated to giving mathematically precise definitions of what a computer is.

When we are working with decision problems, the output must be either ACCEPT or REJECT. When a machine runs an algorithm, it does one of three things:

1. Output ACCEPT
2. Output REJECT
3. Loop forever

Definition 3.3. We say a machine M **decides** a language L if for all w

- M outputs ACCEPT if and only if $w \in L$
  - If $w \in L$, then M outputs ACCEPT
  - If $w \notin L$, then M outputs REJECT
- M *never* loops

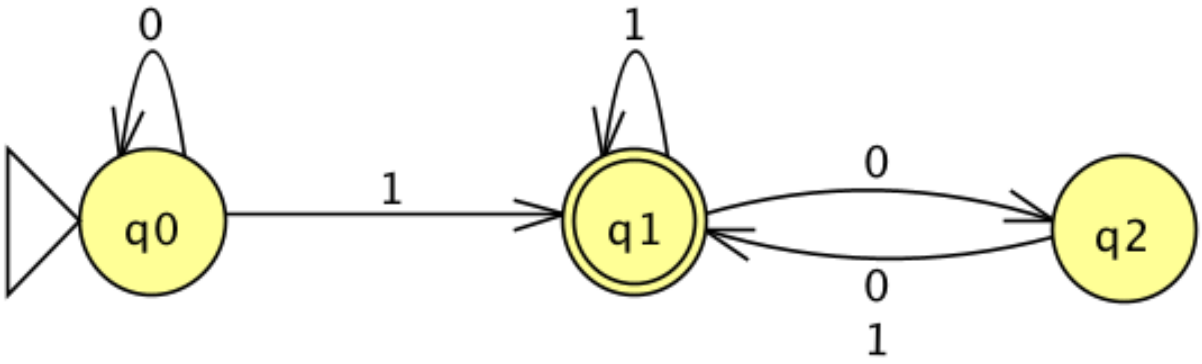

Figure 1: **Example DFA 1**

Definition 3.4. We say a machine M **recognizes** a language L if for all w

- M outputs ACCEPT if and only if $w \in L$
  - If $w \in L$ then M outputs ACCEPT
  - If $w \notin L$, then M *does not* output ACCEPT.
- M is allowed to loop if $w \notin L$

4 Deterministic Finite Automata

In this section we will describe a very simple model of a computer. We will see some of the languages this model of computation can and cannot recognize.

4.1 Formal Description

Definition 4.1. A **deterministic finite automata (DFA)** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

1. Q is a finite set called the **states**
2. Σ is a finite set called the **alphabet**
3. $\delta : Q \times \Sigma \to Q$ is the **transition function**
4. $q_s \in Q$ is the (unique) **start state**
5. $F \subseteq Q$ is the set of **accept states**

Example 4.1. Consider the DFA in Figure 1 The formal description of this DFA is as follows:

1. $Q = \{q_0, q_1, q_2\}$
2. $\Sigma = \{0, 1\}$
3. We can write the transition function in two ways. First, we can write it explicitly
   - $\delta(q_0, 0) = q_0$
   - $\delta(q_0, 1) = q_1$
   - $\delta(q_1, 0) = q_2$
   - $\delta(q_1, 1) = q_1$
   - $\delta(q_2, 0) = q_1$
   - $\delta(q_2, 1) = q_1$

   Alternately we can write it as a table

	0	1
q_0	q_0	q_1
q_1	q_2	q_1
q_2	q_1	q_1

4. $q_s = q_1$
5. $F = \{q_2\}$

Some technical notes:

- Q, Σ, F are all *sets*
- q_s is a single state, i.e. a single element of Q
- δ is defined for every combination of a state and a symbol

4.2 Computation on a DFA

A DFA performs computation by reading each symbol of the input one by one, and moving to a new state upon reading each symbol. Formally, the DFA starts in state $q = q_s$. Let $w = w_1 w_2 \dots w_n$ be the input. It reads each symbol w_i in order. For each w_i, it uses the transition function and computes $\delta(q, w_i)$. Based on this, it transitions to a new state, i.e. $q \leftarrow \delta(q, w_i$. After reading every symbol, the DFA checks if it is in an accept state. If $q \in F$, the DFA outputs ACCEPT; otherwise, if $q \notin F$, the DFA outputs REJECT.

4.2.1 Accepting Computation

Definition 4.2. Let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automaton and let $w = w_1 w_2 \dots w_n$ be a string where each w_i is a member of the alphabet Σ. Then M **accepts** w if and only if a sequence of states $r_0 r_1 r_2 \dots r_n \in Q$ exists with three conditions:

- $r_0 = q_0$
- $\delta(r_i, w_{i+1}) = r_{i+1}$ for $i = 0, 1, \dots, n-1$
- $r_n \in F$

Definition 4.3. M **recognizes** a language L if and only if L = {w | M accepts w}. Note that a DFA cannot ever loop forever, since it stops after reading the last symbol. Thus if a DFA recognizes a language, it also decides it.

4.3 Exercises

1. Let $\Sigma = \{0, 1\}$. Design a DFA to recognize L = {w | w starts and ends with the same symbol}

2. Let $\Sigma = \{+, 1, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$. Design a DFA to recognize the language L = {w | w is a valid integer literal}

3. Let $\Sigma = \{a, b\}$. Design a DFA that recognizes {w = | w ends in bba}

4. Let $\Sigma = \{a, b, \ldots, z\}$. Design a DFA to recognize a string with *exactly* two a's.

5. Design a DFA to recognize all strings that *do not* contain *exactly* two a's.

6. First, Let w^r be the reversal of a string w. Let L be a formal language; define $L^r = \{w^r | w \in L\}$.

Let $\Sigma_3 = \{0, 1\}^3$ i.e. $\begin{bmatrix}0\\0\\0\end{bmatrix}, \begin{bmatrix}0\\0\\1\end{bmatrix}, \ldots, \begin{bmatrix}1\\1\\1\end{bmatrix}$.

Let $B = \{w \in \Sigma_3^* | \text{the top row} + \text{middle row} = \text{the bottom row}\}$

Design a DFA to recognize B^r. Remember, the DFA reads a column vector at every step

4.4 Regular Languages

Definition 4.4. A formal language L is a **regular language** if and only if some DFA D recognizes L. To show that a language is regular, we must construct a DFA to recognize L.

Definition 4.5. Let $L \subseteq \Sigma^*$ be a formal language on an alphabet Σ. The **complement** of L is

$$L^c = \{w | w \in \Sigma^*, w \notin L\} = \Sigma^* \setminus L$$

i.e. all strings not in L

Lemma 4.1. Regular languages are closed under complement. That is, if L is regular, then L^c is regular

Proof. If L is regular, then some DFA D recognizes L - that is, if $w \in L$ then D accepts w, and if $w \notin L$ then D rejects w. To show that L^c is regular, we must design a DFA D^c to recognize L^c. We simply take D and flip the accept/reject states, so that the DFA rejects all strings in L and accepts all strings not in L.

Formally, Let $D = (Q, \Sigma, \delta, q_0, F)$. We create a DFA D^c to recognize L^c. The formal description of $D^c = (Q^c, \Sigma^c, \delta^c, q_0^c, F^c)$ is as follows:

1. $Q^c = Q$ i.e. same states
2. $\Sigma^c = \Sigma$ i.e. same alphabet
3. For all $q \in Q, w \in \Sigma^c$, $\delta^c(q, w) = \delta(q, w)$ i.e. same transition function
4. $q_0^c = q_0$ i.e. same start state
5. $F^c = Q \setminus F$ i.e. flip the reject states to accept states

By construction, if D rejects w then D^c accepts w because the final state flips from a reject state to an accept state; similarly if D accepts w then D^c rejects w. □

Definition 4.6. Let A and B be regular languages. The **perfect shuffle** of A and B is

$$\text{PERFECT-SHUFFLE}(A, B) = \{w = a_1 b_1 a_2 b_2 \ldots a_n b_n | a_1 a_2 \ldots a_n \in A, b_1 b_2 \ldots b_n \in B\}$$

i.e. the odd characters form a string in A and the even characters form a string in B.

Lemma 4.2. Regular languages are closed under the perfect shuffle operation. That is, if A and B are regular, then PERFECT-SHUFFLE(A, B) is regular

Proof. We know that there exists DFAs $D_A = (Q_A, \Sigma_A, \delta_A, q_{s_A}, F_A)$ and $D_B = (Q_B, \Sigma_B, \delta_B, q_{s_B}, F_B)$. We will construct a DFA D to check if the even charaters form a string that is accepted by D_A and if the even characters form a string that is accepted by D_B. We will use the technique of running two DFAs in alternation.

Every state in D is a 3-tuple corresponding to a combination of a state in A, a state in B, and a variable that keeps track of whose 'turn 'it is to read a character. Whenever the machine reads a character, it checks whose turn it is; that machine whose turn it is reads the character and updates its state while the other machine stays in the same state. The machine then switches value of the turn variable. After reading all the characters, the machine checks if both of the smaller machines have reached accept states, and if it is A's turn to read a character (both machines should read the same number of characters). If so it accepts, otherwise it rejects.

The formal description of $D = (Q, \Sigma, \delta, q_s, F)$ is as follows:

1. $Q = Q_A \times Q_B \times \{A, B\}$ Every state in D contains three elements: a state from D_A, a state from D_B, and a 'turn 'counter with value A or B.
2. $\Sigma = \Sigma_A \cup \Sigma_B$
3. $$\delta((q_A, q_B, A), w) = (\delta_A(q_A, w), q_B, B)$$ $$\delta((q_A, q_B, B), w) = (q_A, \delta_B(q_B, w), A)$$ If it is A's turn, the machine computes the transition function for A then switches to B. Otherwise, it computes the transition function for B and switches to A.
4. $q_s = (q_{s_A}, q_{s_B}, A)$ i.e. the machine starts in the two start states for D_A and D_B, and at the start it is A's turn to read a character.
5. $F = F_A \times F_B \times \{A\}$ i.e. the machine accepts if both D_A and D_B would be in an accept state, and if after reading the characters it is A's turn to read.

□

4.5 Regular operations

Definition 4.7. Next, let L_1 and L_2 be formal languages. The three **regular operations** are the following:

- The **union** of L_1 and L_2 is
$$L_1 \cup L_2 = \{w | w \in L_1 \vee w \in L_2\}$$
i.e. all strings that are in either one of the two languages.
- The **concatenation** of L_1 and L_2 is
$$L_1 \circ L_2 = \{w_1 w_2 | w_1 \in L_1, w_2 \in L_2\}$$
i.e. a string in L_1 followed by a string in L_2
- The **Kleene star** of L_1 is
$$L_1^* = \{\epsilon\} \cup \{w_1 w_2 w_3 \dots w_n | w_i \in L_1\}$$
i.e. 0 or more consecutive strings, all of which are in L L_1

Are regular languages closed under the regular operations?

Lemma 4.3. Regular languages are closed under union. That is, if L_1 and L_2 are regular, then $L_1 \cup L_2$ is regular.

Proof. We know that L_1, L_2 are both regular, so there exist DFAs $D_1 = (Q_1, \Sigma_1, \delta_1, q_{s_1}, F_1)$ and $D_2 = (Q_2, \Sigma_2, \delta_2, q_{s_2}, F_2)$ that recognize L_1 and L_2, respectively. To show that $L_1 \cup L_2$ is regular, we must show that there is a DFA D that recognizes $L_1 \cup L_2$. To construct D, we will use the technique of running two DFAs in parallel. Every state in D will be a combination of a state in D_1 and a state in D_2. Whenever D reads a character, it simultaneously computes the transition function for D_1 and D_2. This produce a state from D_1 and a state from D_2; D transitions to the state corresponding to that combination of states. The machine accepts if after reading the string, at least one of the smaller machines has moved to an accept state.

Formally, $D = (Q, \Sigma, \delta, q_s, F)$ is defined as follows:

- $Q = Q_1 \times Q_2 = \{(q_1, q_2) | q_1 \in Q_1, q_2 \in Q_2\}$ i.e. every state in Q contains a pair of states, one from D_1 and one from D_2
- $\Sigma = \Sigma_1 \cup \Sigma_2$ i.e. the union of the two alphabets
- $\delta((q_1, q_2), w) = (\delta_1(q_1, w), \delta_2(q_2, w))$, i.e. the transition function takes as input a pair of states (q_1, q_2) and a symbol; it simultaneously computes the transition functions from D_1 and D_2, and outputs a new pair of states. It computes where D_1 should transition if it is in state q_1 and reads w, and it computes where D_2 should transition if it is in state q_2 and reads w. In this way, the machine runs D_1 and D_2 *in parallel.*
- $q_s = (q_{s_1}, q_{s_2})$ i.e. when D runs D_1 and D_2 in parallel, it starts with D_1 and D_2 in their respective start states
- $F = \{(f_1, f_2) | f_1 \in F_1 \vee f_2 \in F_2\}$ i.e. D accepts if after running D_1 and D_2 in parallel, either machine ends up in accept states

By construction, our machine D accepts if at least one of D_1 or D_2 accepts. □

Corollary 4.1. Regular languages are closed under intersection. That is, if L_1 and L_2 are regular, then $L_1 \cap L_2$ is regular.

Proof. We can prove this in two ways. The first way is to use the technique of the Cartesian product construction. However, rather than accepting if just one of the smaller machines accept, we check that *both* machines accept.

Formally, let DFAs $D_1 = (Q_1, \Sigma_1, \delta_1, q_{s_1}, F_1)$ and $D_2 = (Q_2, \Sigma_2, \delta_2, q_{s_2}, F_2)$ recognize L_1 and L_2, respectively. $D = (Q, \Sigma, \delta, q_s, F)$ is defined as follows:

- $Q = Q_1 \times Q_2 = \{(q_1, q_2) | q_1 \in Q_1, q_2 \in Q_2\}$ i.e. every state in Q contains a pair of states, one from D_1 and one from D_2
- $\Sigma = \Sigma_1 \cup \Sigma_2$ i.e. the union of the two alphabets
- $\delta((q_1, q_2), w) = (\delta_1(q_1, w), \delta_2(q_2, w))$, i.e. the transition function takes as input a pair of states (q_1, q_2) and a symbol; it simultaneously computes the transition functions from D_1 and D_2, and outputs a new pair of states. It computes where D_1 should transition if it is in state q_1 and reads w, and it computes where D_2 should transition if it is in state q_2 and reads w. In this way, the machine runs D_1 and D_2 *in parallel.*
- $q_s = (q_{s_1}, q_{s_2})$ i.e. when D runs D_1 and D_2 in parallel, it starts with D_1 and D_2 in their respective start states
- $F = F_1 \times F_2$ i.e. D accepts if after running D_1 and D_2 in parallel, both machines end up in accept states

By construction, our machine D accepts if at least one of D_1 or D_2 accepts.

We may also prove it using the technique of expressing intersection in terms of other operations. We already know that regular languages are closed under union and complement. We then note that by De Morgan's laws

$$L_1 \cap L_2 = (L_1^c \cup L_2^c)^c$$

Because L_1 and L_2 are both regular, the expression on the right hand side is regular, thus the expression on the left hand side is regular. □

Lemma. Regular languages are closed under concatenation. That is, if L_1, L_2 are regular then $L_1 \circ L_2$ is regular.

Again, we note that L_1 and L_2 are recognized by machines D_1 and D_2. However, we can't just run D_1 and D_2 in parallel; we have to first run D_1, and then run D_2. However, how do we know when stop running D_1 and start running D_2? This will be very tricky to do with a DFA. Thankfully, there is another type of machine that may help us.

5 Nondeterministic Finite Automata

In this section we will describe a model of computation that extends some of the abilities of a DFA.

5.1 Formal Description

Definition 5.1. Let S be a set. The **power set of S**, denoted $\mathcal{P}(S)$ is the set of all subsets of S, i.e.

$$\mathcal{P}(S) = \{S' | S' \subseteq S\}$$

Example 5.1. Let $S = \{1, 2, 3\}$. Then $\mathcal{P}(S) = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$

Definition 5.2. A **Nonteterministic finite automata (NFA)** is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

1. Q is a finite set of states
2. Σ us a finite alphabet
3. $\delta : Q \times \Sigma_\epsilon \to \mathcal{P}(Q)$ is the transition function, where $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$
4. $q_s \in Q$ is the start state
5. $F \subseteq Q$ is the set of accept states

Some notes:

- In a DFA, the transition function takes as input a single state and a single character, and outputs a *single state*. The transition function takes as input a single state, and a single character, and it outputs a *set of states*.
- In a DFA, every state has exactly one transition for every symbol. In an NFA, the same state may have multiple possible transitions for the same symbol
- In addition to the alphabet Σ, there may be transitions with the label ϵ. This essentially lets the NFA transition without actually reading a character.

Example 5.2. Consider the NFA in Figure 2 The formal description is as follows:

1. $Q = \{q_0, q_1, q_2, q_3\}$

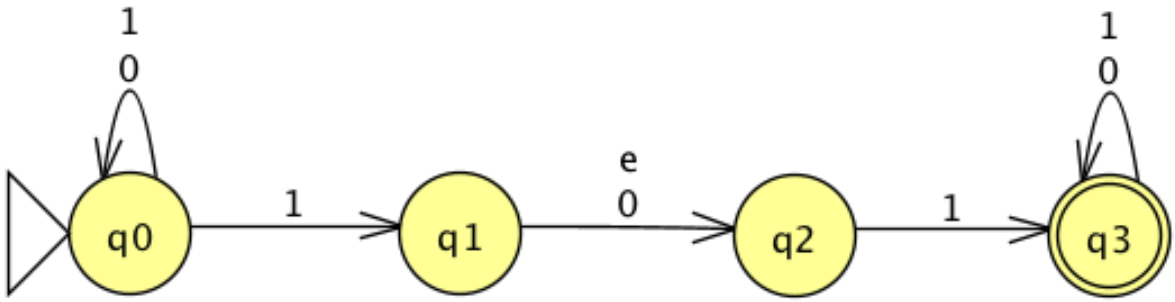

Figure 2: **Example NFA 1**

2. $\Sigma = \{0, 1\}$
3. We can describe the transition function in two ways. First, we'll describe it explicitly

   - $\delta(q_0, 0) = \{q_0\}$
   - $\delta(q_0, 1) = \{q_0, q_1\}$
   - $\delta(q_0, \epsilon) = \emptyset$
   - $\delta(q_1, 0) = \{q_3\}$
   - $\delta(q_1, 1) = \emptyset$
   - $\delta(q_1, \epsilon) = \{q_3\}$
   - $\delta(q_2, 0) = \emptyset$
   - $\delta(q_2, 1) = \{q_3\}$
   - $\delta(q_2, \epsilon) = \emptyset$
   - $\delta(q_3, 0) = \{q_3\}$
   - $\delta(q_3, 1) = \{q_3\}$
   - $\delta(q_3, \epsilon) = \emptyset$

   We'll also describe it in a table

	0	1	ϵ
q_0	$\{q_0\}$	$\{q_0, q_1\}$	$\emptyset$
q_1	$\{q_2\}$	$\emptyset$	$\{q_2\}$
q_2	$\emptyset$	$\{q_3\}$	$\emptyset$
q_3	$\{q_3\}$	$\{q_3\}$	$\emptyset$

4. $q_s = q_0$
5. $F = \{q_3\}$

5.2 Computation on an NFA

An NFA processes a string in the same way as a DFA. It reads one character at a time and moves from one state to state. However, when the NFA reads a character, there may be multiple different available

transitions. The NFA can 'choose' which way to go. Thus, an NFA could potentially read the same string and end up in completely different states at the end! Furthermore, the NFA may make choices and eventually reach a point where it has no available transitions, and the computation dies. How do we define what means for an NFA to accept a string when the NFA can behave in several different ways on the same string?

5.3 Accepting computation

Definition 5.3. An NFA accepts a string w if we can write w as $w = y_1 y_2 \dots y_m$, where each y_i is a member of Σ_ϵ, and there exists as sequence of states $r_0 r_1, \dots, r_m$ such that

1. $r_i \in Q$ for all i
2. $r_0 = q_0$
3. $r_{i+1} \in \delta(r_i, y_{i+1}$ for all $i = 0, \dots, m-1$
4. $r_m \in F$

Conditions 1, 2, and 4 are straightforward. Condition 3 states that each r_{i+1} is one of the (possibly many) allowable states following r_i upon reading character y_{i+1}. Some additional notes:

- We may re-write w by inserting any number of ϵ transitions
- We only need there to exist at least one accepting computation path. Even if all other paths lead to a reject state or die, just one accepting path is sufficient.

5.4 Exercises

1. Design an NFA with four states to recognize {w | w ends with bba}

2. Design an NFA with four states to recognize {w | w contains bba}

6 Equivalence of NFAs and DFAs

Theorem 6.1. A language L can be recognized by an NFA if and only if L is recognized by a DFA

Proof. **Proof idea:** If an NFA N recognizes L, we will construct a DFA D to recognize L. The DFA D will simulate all the possible computation paths that N could take on a string w. D will have a state for every possible set of states that N could be in at any time. The transition function will take in a set of states R as input, and a character a; R represents all the possible states that R could be in at the current point in the computation. The transition function will output all the possible states that the machine could transition to upon reading a. After reading all characters in w, it will check if N could *possibly* be in an accept state.

($\Rightarrow$) To prove the forward direction, we will show that if a language L can be recognized by a DFA D, then L can be recognized by an NFA N. We simply note that since D is a DFA, D *is* an NFA that simply chooses not to have any ϵ transitions. Thus, D is an NFA that recognizes L.

($\Leftarrow$) To prove the backwards direction, we will show that if a language L can be recognized by an NFA N, then L can be recognized by a DFA D. To do this, we will use the technique of the power set construction. Let $N = (Q_n, \Sigma_n, \delta_n, q_{s_n}, F_n)$. First, let's consider the case where there are no ϵ transitions. We will define a DFA $D = (Q_d, \Sigma_d, \delta_d, q_{s_d}, F_d)$ as follows:

1. $Q_d = \mathcal{P}(Q_n)$ i.e. the power set of the states in N
2. $\Sigma_d = \Sigma_n \setminus \{\epsilon\}$ i.e. same alphabet with no ϵ transitions

3. $\delta_d(R, a) = \{q \in Q_s | q \in \delta_n(r, a) \text{ for some } r \in R\}$

   If R is a state of D, it is a set of states of N. When D reads a symbol in a state R, it shows where a takes each state in R. Because each state may go to a set of states, we take the union of all these sets. We could also write this as

$$\delta_d(R, a) = \bigcup_{r \in R} \delta_n(r, a)$$

4. $q_{s_d} = \{q_{s_n}\}$
5. $F_d = \{R \in Q_d | R \text{ contains an accept state of } N\}$

   The machine accepts if one of the possible states that N could be in at that point is an accept state.

Now let's consider ϵ transitions. For any state $R \in Q_d$, define $\mathcal{E}(R)$ to be the collection of states that can be reached from members of R by only going along ϵ transitions, including members of R themselves.

Formally, let $\mathcal{E}(R) = \{q | q \text{ can be reached from R by travelling along 0 or more } \epsilon \text{ transitions}\}$

We modify the transition function as follows:

$$\delta_d(R, a) = \{q \in Q | q \in \mathcal{E}(\delta_n(r, a)) \text{ for } r \in R\}$$

Alternately we could write it as

$$\delta_d(R, a) = \bigcup_{r \in R} \mathcal{E}(\delta_n(r, a))$$

We also modify the start state to be $q_{s_d} = \mathcal{E}(\{q_{s_n}\})$ □

Corollary 6.1. L is regular if and only if L is recognized by an NFA

Proof. By definition, L is regular if and only if L is recognized by a DFA. By theorem 6.1, L is recognized by a DFA if and only if L is recognized by an NFA. □

Example 6.1.

Consider the example in Figure 3. We will convert this NFA to a DFA. The formal description is as follows:

1. $Q = \{\emptyset, \{1\}, \{2\}, \{3\}, \{1, 2\}, \{1, 3\}, \{2, 3\}, \{1, 2, 3\}\}$
2. $\Sigma = \{a, b\}$
3. We will write the transition function as a table

	a	b
$\emptyset$	$\emptyset$	$\emptyset$
$\{1\}$	$\emptyset$	$\{2\}$
$\{2\}$	$\{2, 3\}$	$\{3\}$
$\{3\}$	$\{1, 3\}$	$\emptyset$
$\{1, 2\}$	$\{2, 3\}$	$\{2, 3\}$
$\{1, 3\}$	$\{1, 3\}$	$\{2\}$
$\{2, 3\}$	$\{1, 2, 3\}$	$\{3\}$
$\{1, 2, 3\}$	$\{1, 2, 3\}$	$\{2, 3\}$

4. $q_s = \mathcal{E}(\{1\}) = \{1, 3\}$
5. $F = \{\{1\}, \{1, 2\}, \{1, 3\}, \{1, 2, 3\}\}$

The final DFA shown in Figure 4.

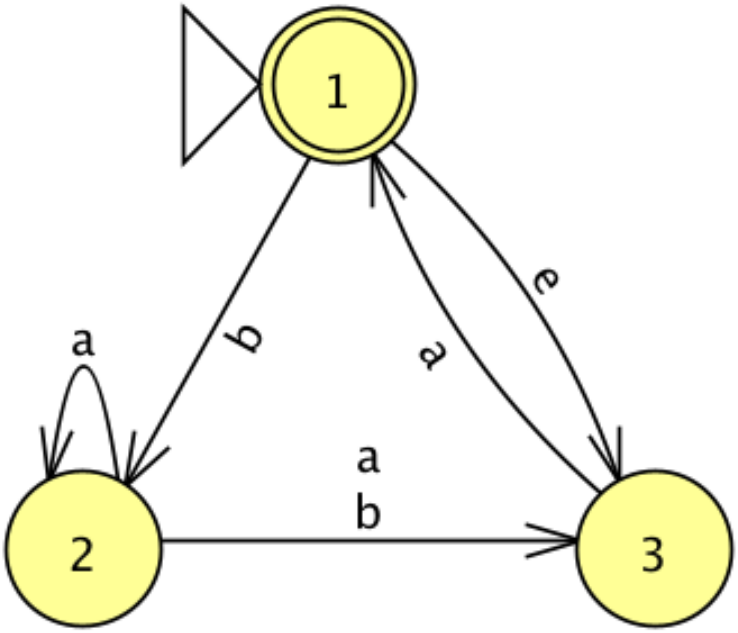

Figure 3: **Example NFA 2**

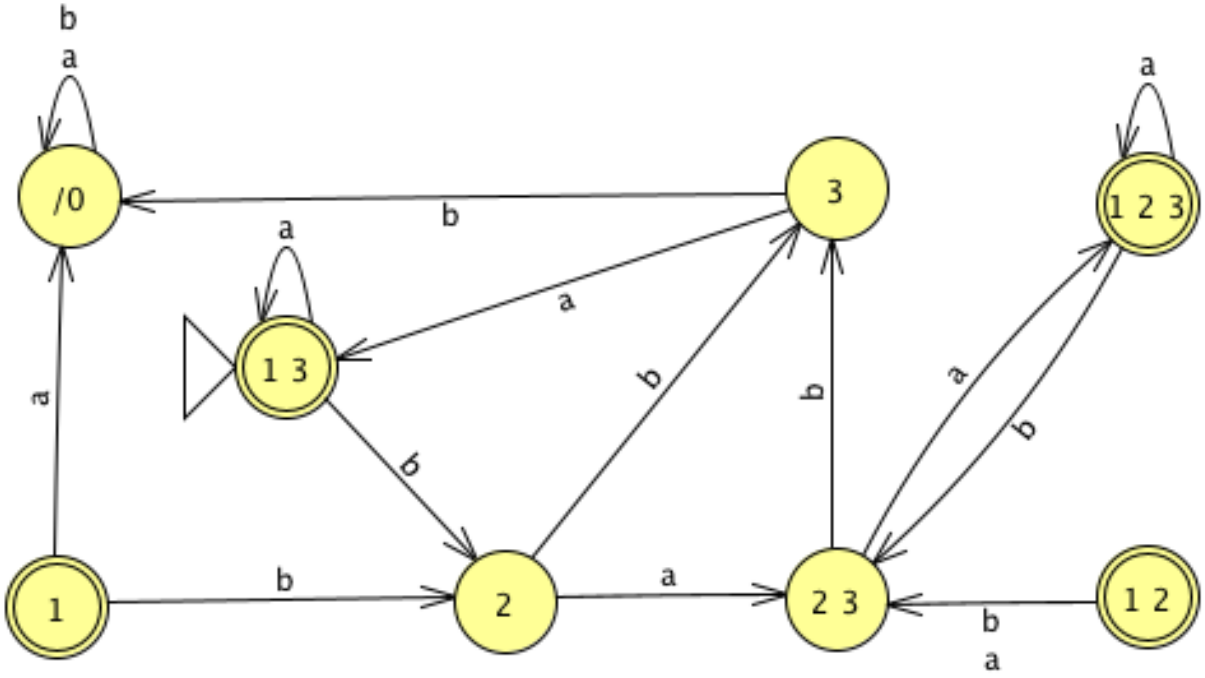

Figure 4: **NFA to DFA conversion**

7 Closure Properties of Regular Languages

Theorem 7.1 (Kleene's Theorem)**.** The regular languages are closed under the regular operations

Proof. Suppose we have languages L_1, L_2 which are recognized by NFAs $N_1 = (Q_1, \Sigma, \delta, q_{s_1}, F_1), N_2 = (Q_2, \Sigma, \delta_2, q_{s_2}, F_2)$, respectively. For simplicity we'll assume the alphabets are the same for both languages. We will construct an NFA $N = (Q, \Sigma, \delta, q_s, F)$ as follows

Union: Combine N_1 and N_2 into a single NFA N. The machine nondeterministically guesses whether to run w through N_1 or N_2, and accepts if either machine accepts.

Formally, create a new start state q_s, and modify the transition function so that $\delta(q_s, \epsilon) = \{q_{s_1}, q_{s_2}\}$

Concatenation: Combine N_1 and N_2 into a single NFA N. The machine nondeterministically guesses how to split up w into two substrings. It reads the first substring through N_1, transitions to N_2, and reads the second substring through N_2. If both machines accept the substrings that they read, N accepts.

Formally, let F_1 be the accept states for N_1 and q_{s_2} be the start state for N_2. Modify the transition function so that for all $f \in F_1$, $q_{s_2} \in \delta(f, \epsilon)$. The start state for N is the start state for N_1. The accept states for N are the accept states for N_2.

Kleene Star: Start with N_1. The machine nondeterministically guesses how to split w into several substrings. It reads each substring through N_1; when each substring reaches an accept state, N loops back to the start state to read through the next substring.

Formally, let q_{s_1} be the start state for N_1, and let F_1 be the accept states. Create a new start state q_s. Modify the transition function so $\delta(q_s, \epsilon) = \{q_{s_1}\}$, and for all $f \in F_1$, $q_{s_1} \in \delta(f, \epsilon)$. The accept states are $F_1 \cup \{q_s\}$; we make q_s an accept state so that the machine accepts ϵ. □