Introduction to automata theory

Jyun-Ao Lin

iFIRST & CSIE, NTUT

Credits

A large part of this slide is based on the previous FLOLAC course of Bow-Yaw Wang[1]

[1] basically follows [Sip13]

Why study automata theory?

a Turing award winning paper

M. O. Rabin*
D. Scott†

Finite Automata and Their Decision Problems‡

Abstract: Finite automata are considered in this paper as instruments for classifying finite tapes. Each one-tape automaton defines a set of tapes, a two-tape automaton defines a set of pairs of tapes, et cetera. The structure of the defined sets is studied. Various generalizations of the notion of an automaton are introduced and their relation to the classical automata is determined. Some decision problems concerning automata are shown to be solvable by effective algorithms; others turn out to be unsolvable by algorithms.

Introduction

Turing machines are widely considered to be the abstract prototype of digital computers; workers in the field, however, have felt more and more that the notion of a Turing machine is too general to serve as an accurate model of

a method of viewing automata but have retained throughout a machine-like formalism that permits direct comparison with Turing machines. A neat form of the definition of automata has been used by Burks and Wang[1]

for people who love automata

- string processing/tokenization/pattern matching: lexical analyzer, parser, search engine, XML/regex processing, etc
- verification/model checking: Spin, UAutomizer, RMC, AutoQ, etc
- system design/modeling/simulation: transition system, proof system, network protocol, etc
- logic/specification language: LTL, MSO, WSkS, etc
- SMT: string solver, Presburger arithmetic, LIA, etc
- synthesis
- game theory
- complexity theory
- ...etc
- and more next week

in short, a bit old but still relevant

Table of contents

Deterministic finite automata

schematic of deterministic finite automata

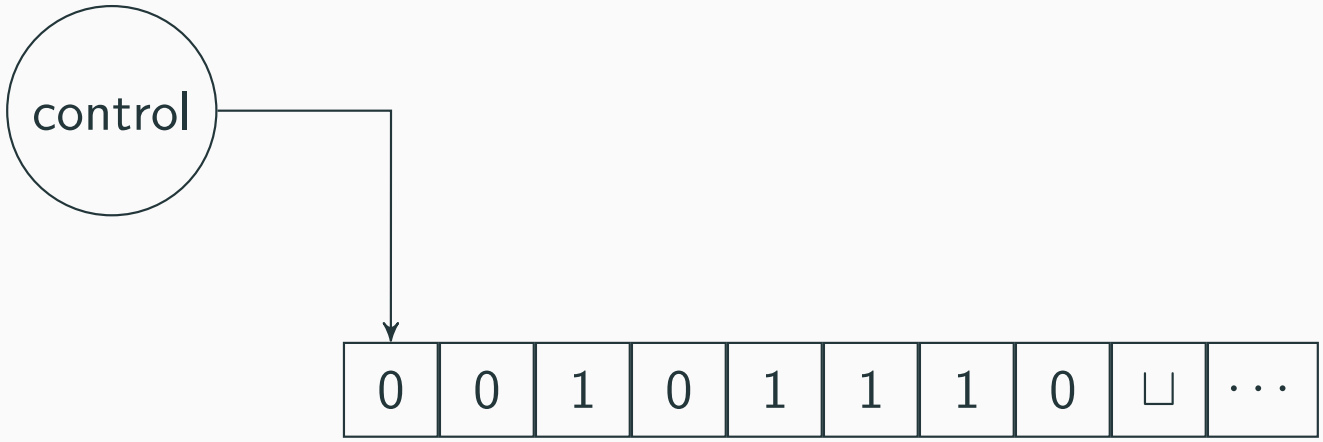


Figure 1: Schematic of finite automata

- a finite automation has a finite set of control *states*.
- a finite automation reads input symbols of from left to right.
- a finite automation accepts or rejects an input after reading the input.

finite automation M_1

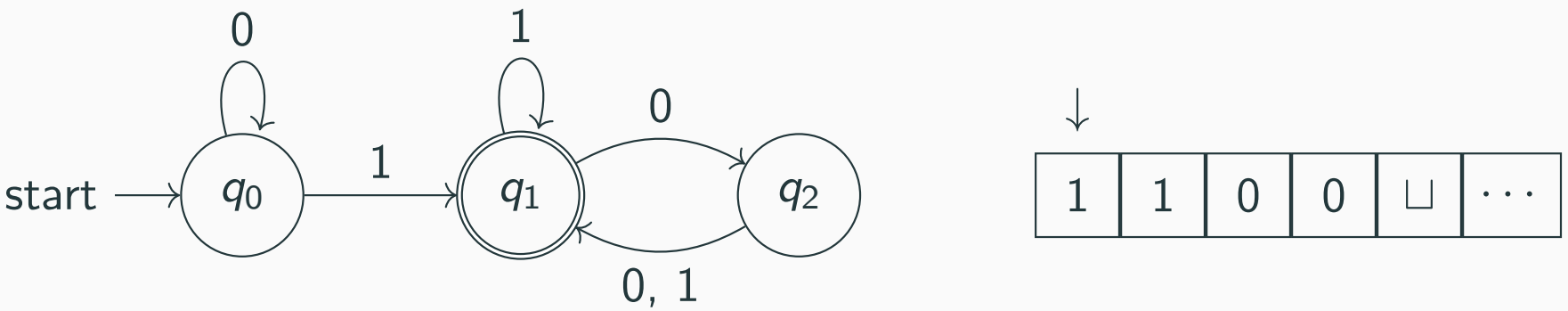
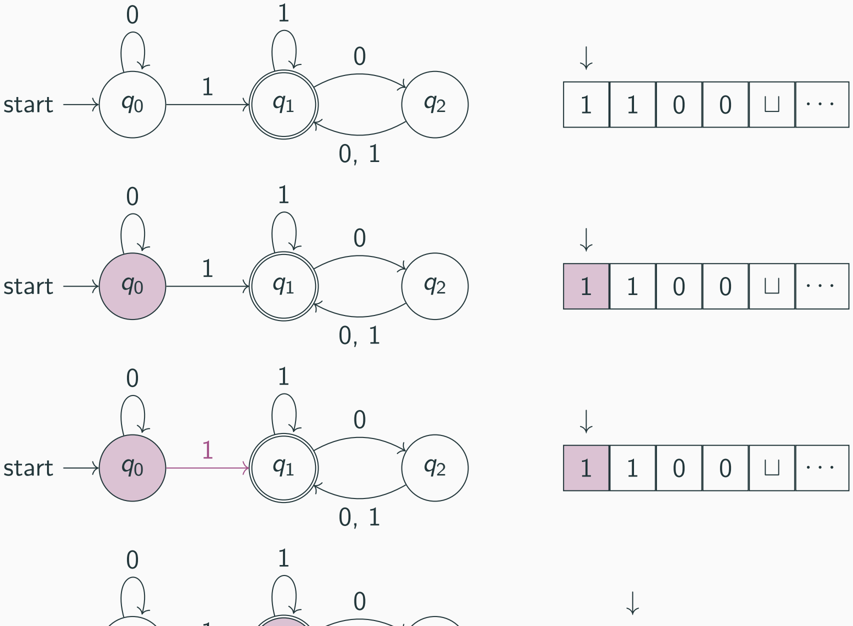


Figure 2: DFA M_1

Fig. 2 shows the state diagram of an automation M_1, which has

- 3 states: q_0, q_1, q_2;
- a start state: q_0;
- an accepting state: q_1;
- 6 transitions: $q_0 \xrightarrow{0} q_0, q_0 \xrightarrow{1} q_1, q_1 \xrightarrow{1} q_1, q_1 \xrightarrow{0} q_2, q_2 \xrightarrow{0} q_1$ and $q_2 \xrightarrow{1} q_1$.

accepting and rejecting strings



DFAs as data structures

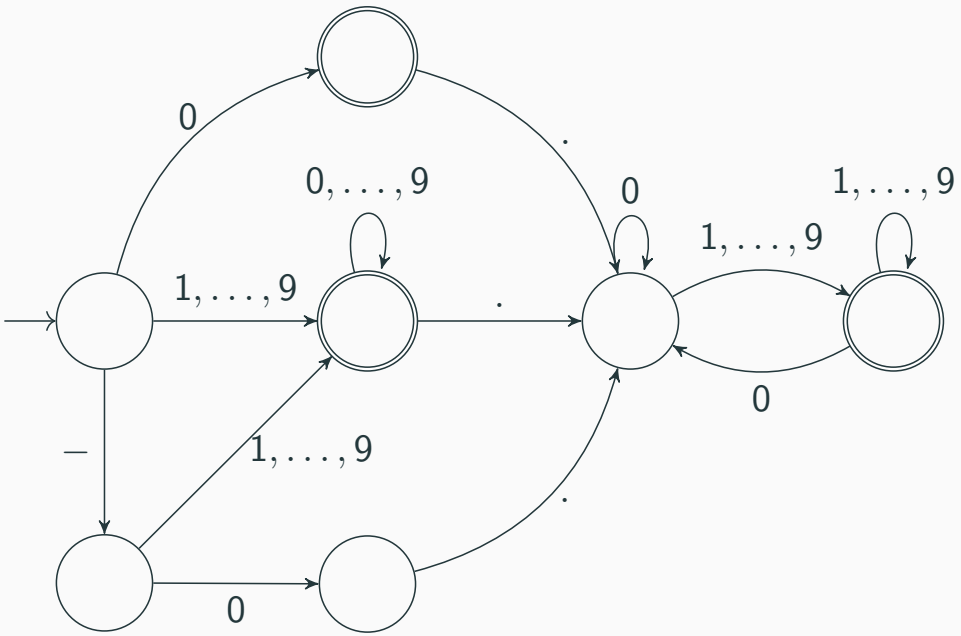


Figure 3: a DFA for decimal numbers

- the DFA on the left recognizes the strings over alphabet $\{-, ., 0, 1, \ldots, 9\}$ that encodes real numbers with a finite decimal part.
- one wishes to exclude $002, -0$ or 3.10000 but accept $37, 10.503$ or -0.2345
- an English description of the correct encoding is rather long...

DFA as data structures

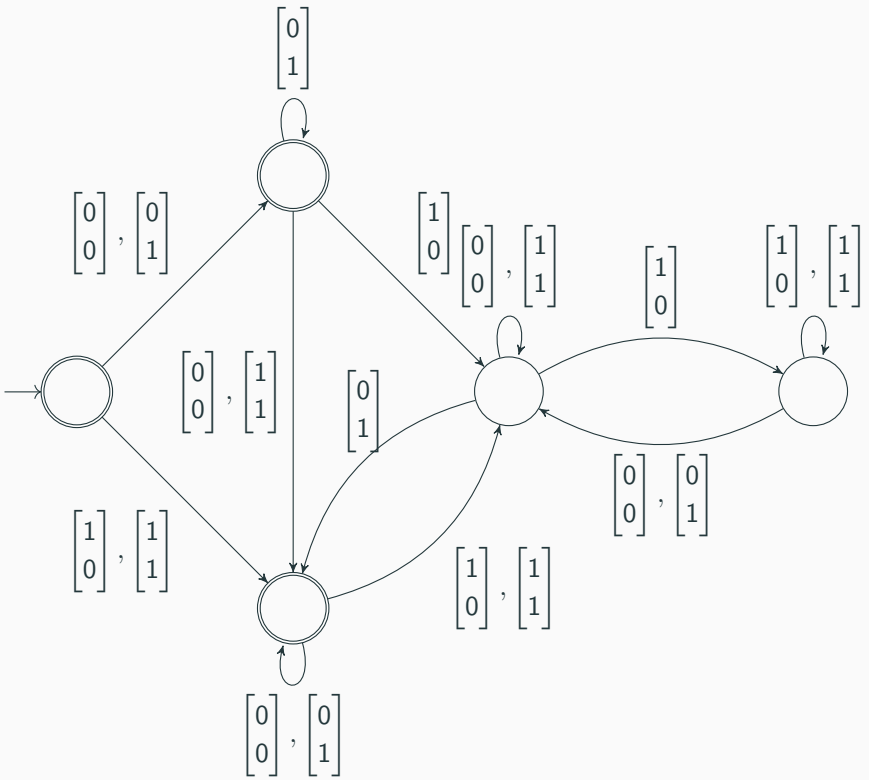


Figure 4: a DFA for the solution of $2x - y \leq 2$

- the inequality $2x - y \leq 2$ has infinitely many integer solutions
- one can encode the solutions as words over alphabet $\{\begin{bmatrix}0\\0\end{bmatrix}, \begin{bmatrix}0\\1\end{bmatrix}, \begin{bmatrix}1\\0\end{bmatrix}, \begin{bmatrix}1\\1\end{bmatrix}\}$ as a DFA
- e.g.

$$\begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}1\\0\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}\begin{bmatrix}0\\1\end{bmatrix}$$

the top row 101100_2 encodes

$1\cdot2^0+0\cdot2^1+1\cdot2^2+1\cdot2^3+0\cdot2^4+0\cdot2^5 = 13$

and the bottom $010011_2 = 50$.

DFA – formal definition

Defn. (DFA)
A deterministic finite automation M is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$, where

- Q is a finite set of states;
- Σ is the set of input symbols, i.e., an alphabet;
- $\delta : Q \times \Sigma \to Q$ is the transition function;
- $q_0 \in Q$ is the start state; and
- $F \subseteq Q$ is the set of accepting or final states.

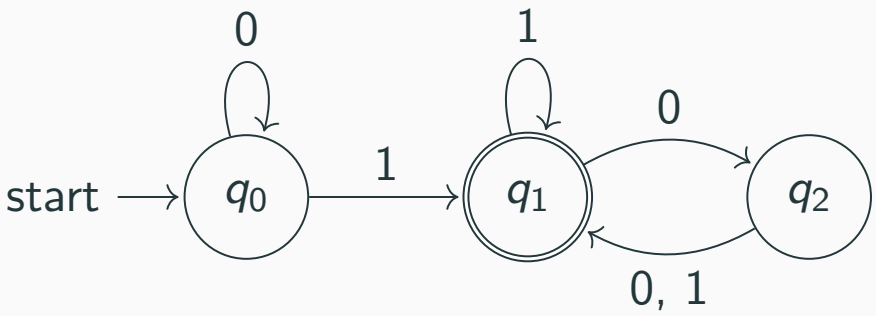
□

informally,

- the set of strings accepted by a DFA M is called the language of M, written as $L(M)$
- we also say M recognizes or accepts $L(M)$

M_1 – formal definition

the DFA $M_1 = (Q, \Sigma, \delta, q_1, F)$ consists of



- $Q = \{q_0, q_1, q_2\}$;
- $\Sigma = \{0, 1\}$;
- q_0 is the start state;
- $F = \{q_1\}$; and
- $\delta : Q \times \Sigma \to Q$ is

	0	1
q_0	q_0	q_1
q_1	q_2	q_1
q_2	q_1	q_1

moreover, we have

$L(M_1) = \{w \mid w$ contains at least one 1 and an even number of 0's follow the last 1.$\}$

DFA M_2

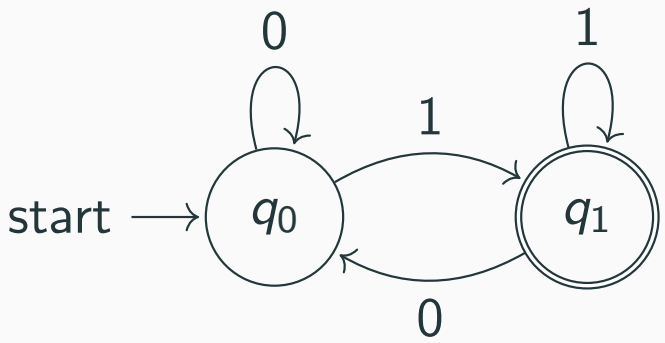


Figure 5: DFA M_2

- Fig. 5 shows $M_2 = (\{q_0, q_1\}, \{0, 1\}, \delta, q_0, \{q_1\})$ where $\delta =$

	0	1
q_0	q_0	q_1
q_1	q_0	q_1

.

- $L(M_2) = \{w \mid w \text{ ends in a } 1\}$.

DFA M_3

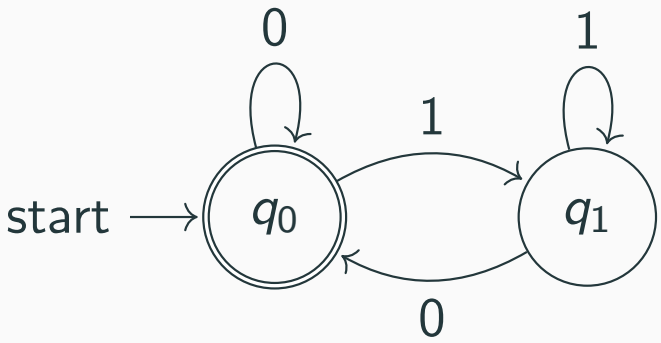


Figure 6: DFA M_3

- Fig. 6 shows $M_3 = (\{q_0, q_1\}, \{0, 1\}, \delta, q_0, \{q_0\})$ with $\delta =$

	0	1
q_0	q_0	q_1
q_1	q_0	q_1

.

- $L(M_3) = \{w \mid w$ ends in a 0 or is the empty string $\epsilon\}$.

DFA M_5

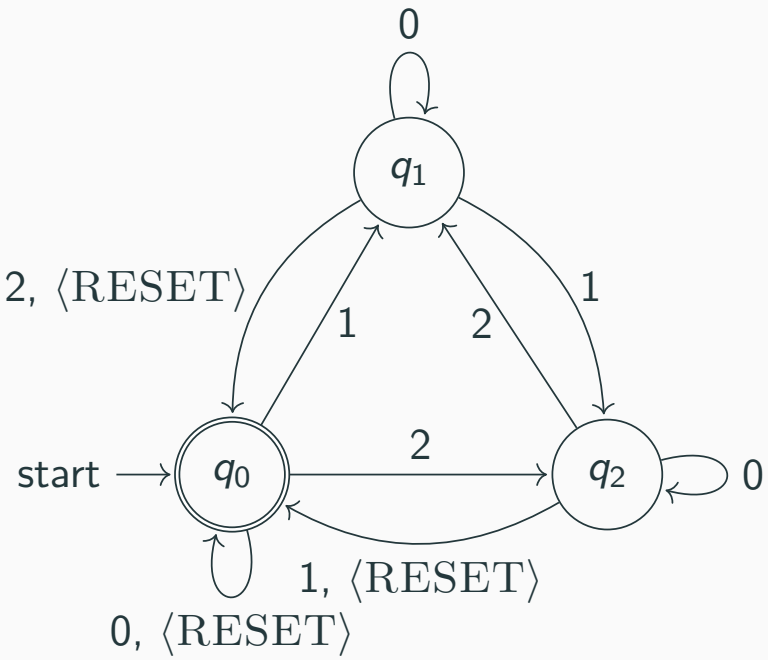


Figure 7: DFA M_5

- Fig. 7 shows $M_5 = (\{q_0, q_1, q_2\}, \{0, 1, 2, \langle \texttt{RESET} \rangle\}, \delta, q_0, \{q_0\})$.
- strings accepted by M_5 are: $\epsilon, 0, 00, 12, 21, 000, 012, 102, 120, 021, 201, 210, 111, 222, \ldots$.
- M_5 computes the sum of input symbols modulo 3. It resets upon the input symbol $\langle \texttt{RESET} \rangle$. Thus M_5 accepts the strings whose sum is a multiple of 3 after $\langle \texttt{RESET} \rangle$.

computation – formal definition

Let $M = (Q, \Sigma, \delta, q_0, F)$ be a DFA.

Defn.

- on an input word $w = a_1 \cdots a_n$ with $a_i \in \Sigma$ for all $i \in [n]$, a run of M on w starting from $p \in Q$ is a sequence:

$$p_0 \; a_1 \; p_1 \; a_2 \; p_2 \; \cdots \; a_n \; p_n$$

  with $p_i \in Q$ for all $i = 0, 1, \ldots, n$, $p_0 = p$ and $\delta(p_j, a_{j+1}) = p_{j+1}$ for all $j = 0, 1, \ldots, n-1$.
- a run of M on w is defined as a run of M on w starting from q_0.
- a run of M on w is called an accepting run if $p_n \in F$.
- a word $w \in \Sigma^*$ is accepted by M if there is an accepting run of M on w.
- the language $L(M)$ of M is the set of words accepted by M.

computation – formal definition

Defn. (regular language)
A language L is regular if there is a DFA M such that $L(M) = L$. □

Remark.

- for every word $w \in L(M)$, there is exactly one run of M on w.
- the empty string ϵ is accepted by M if and only if $q_0 \in F$.

□

boolean operations

Let A and B be languages over an alphabet Σ.

- union: $A \cup B = \{w \mid w \in A \ \vee \ w \in B\}$.
- intersection: $A \cap B = \{w \mid w \in A \ \wedge \ w \in B\}$.
- complementation: $\overline{A} = \{w \mid w \in \Sigma^* \ \wedge \ w \notin A\}$.

Thm.
Regular languages are closed under Boolean operations. □

Thm. (closed under complement)
For every DFA M over an alphabet Σ, there exists a DFA M' over Σ such that $L(M') = \overline{L(M)}$. □

Proof.
Let $M = (Q, \Sigma, \delta, q_0, F)$. Define $M' := (Q, \Sigma, \delta, q_0, F' := Q \setminus F)$. ■

Algorithm for DFA complementation

Algorithm 1 DFA complementation CompDFA(M)

Input DFA $M = (Q, \Sigma, \delta, q_0, F)$

Output DFA $M' = (Q', \Sigma, \delta', q_0', F')$ with $L(M') = \overline{L(M)}$

1: $Q' \leftarrow Q$; $\delta' \leftarrow \delta$; $q_0' \leftarrow q_0$; $F' = \emptyset$
2: **for all** $q \in Q$ **do**
3: **if** $q \notin F$ **then**
4: $F'.\texttt{add}(q)$

Proof detail

we demonstrate here how to write a formal proof.

Proof.
want to show $L(M') = \overline{L(M)} = \Sigma^* \setminus L(M)$:

- $L(M') \subseteq \Sigma^* \setminus L(M)$, i.e., for each $w \in L(M')$ we have $w \in \Sigma^* \setminus L(M)$.
  - for each $w = a_1 \dots a_n \in L(M')$, there is a (unique) run $\rho = q_0 a_1 q_1 a_2 \cdots a_n q_n$ of M' such that $\delta'(q_j, a_{j+1}) = q_{j+1}$ for all $j = 0, 1, \dots, n-1$.
  - w is accepted by $M' \implies q_n \in F' = Q \setminus F$ by construction. We have $q_n \notin F$.
  - Then $w \notin L(M)$ by viewing ρ as a run of M. Therefore $w \in \Sigma^* \setminus L(M) = \overline{L(M)}$.
- $(\Sigma^* \setminus L(M)) \subseteq L(M')$, i.e., for each $w \in \Sigma^* \setminus L(M)$ we have $w \in L(M')$.
  - For each $w = a_1 \dots a_n \in \overline{L(M)}$, there is a (unique) run $\rho' = q_0 a_1 q_1 \dots a_n q_n$ of M such that $\delta(q_j, a_{j+1}) = q_{j+1}$ for all $j = 0, 1, \dots, n-1$.
  - Since $w \notin L(M)$, we have $q_n \notin F$ and hence $q_n \in Q \setminus F$.
  - However, by viewing ρ' as a run of M', ρ' is an accepting run of M' on w.
  - Thus $w \in L(M')$.

boolean operations (cont.)

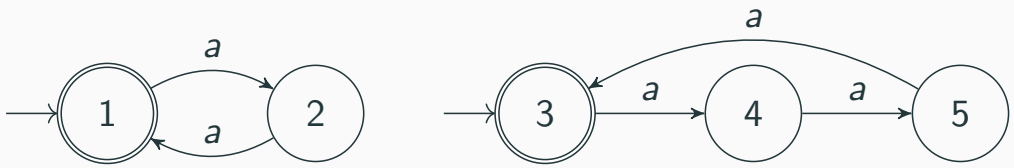
Thm. (closed under intersection)
For every two DFA M_1 and M_2, there is a DFA M' such that $L(M') = L(M_1) \cap L(M_2)$. □

Proof.
Let $M_1 = (Q_1, \Sigma, \delta_1, p_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, p_2, F_2)$. Construct $M' = (Q, \Sigma, \delta, q_0, F)$ as follows:

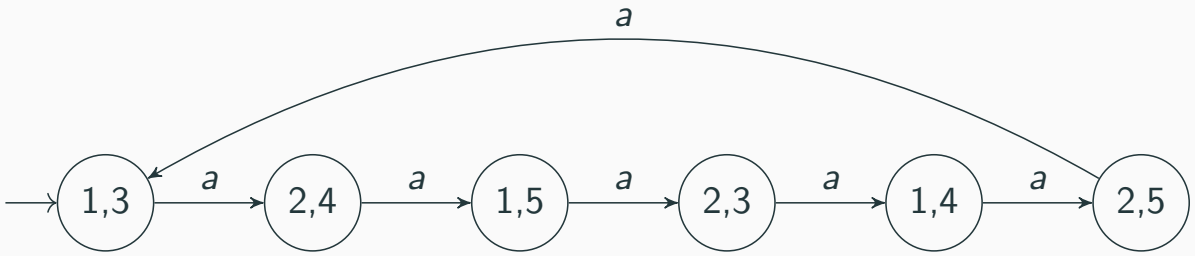
- $Q := Q_1 \times Q_2$;
- $\delta((r_1, r_2), a) := (\delta_1(r_1, a), \delta_2(r_2, a))$;
- $q_0 := (p_1, p_2)$;
- $F := F_1 \times F_2$.

■

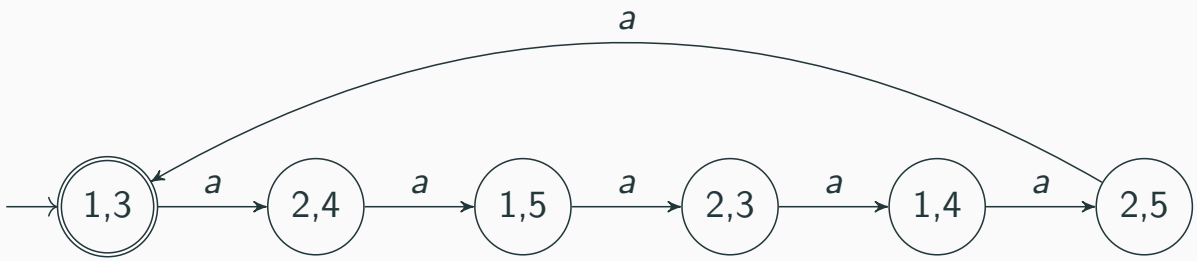
intuition



(a) DFA M_1 and M_2



(b) product DFA $M_1 \times M_2$



(c) DFA for intersection of M_1 and M_2

boolean operations (cont.)

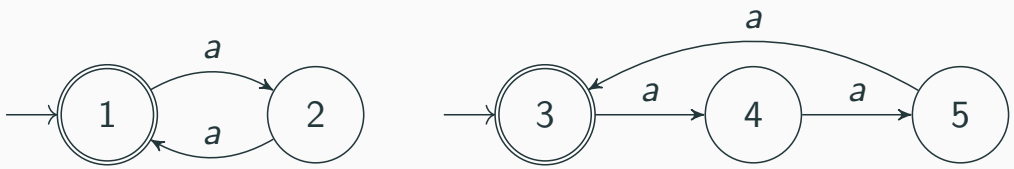
Thm. (closed under union)
For every two DFA M_1 and M_2, there is a DFA M' such that $L(M') = L(M_1) \cup L(M_2)$. □

Proof.
Let $M_1 = (Q_1, \Sigma, \delta_1, p_1, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, p_2, F_2)$. Construct $M' = (Q, \Sigma, \delta, q_0, F)$ as follows:

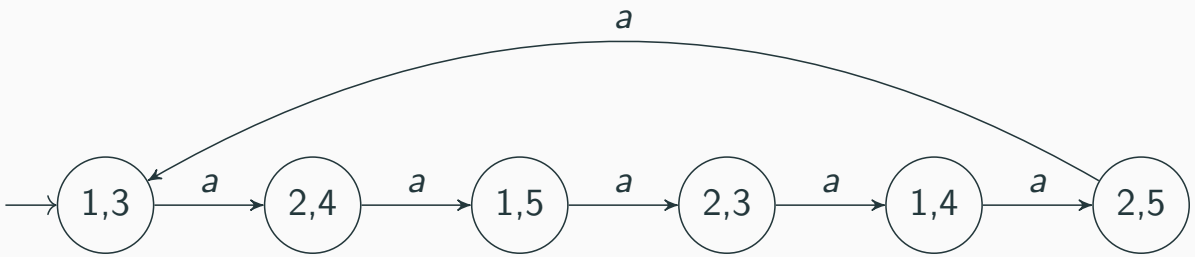
- $Q := Q_1 \times Q_2$;
- $\delta((r_1, r_2), a) := (\delta_1(r_1, a), \delta_2(r_2, a))$;
- $q_0 := (p_1, p_2)$;
- $F := (F_1 \times Q_2) \cup (Q_1 \times F_2) = \{(r_1, r_2) \mid r_1 \in F_1 \vee r_2 \in F_2\}$.

■

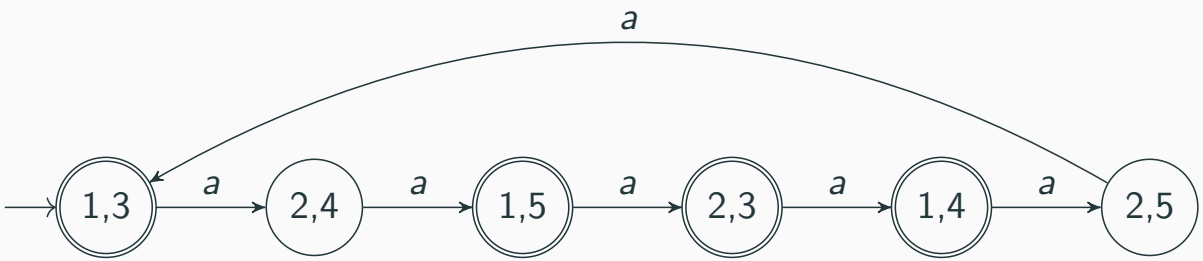
intuition



(a) DFA M_1 and M_2



(b) Product DFA $M_1 \times M_2$



(c) DFA for union of M_1 and M_2

algorithms for boolean combinations

Algorithm 2 Boolean Combination of two DFA BinOp$[\odot](M_1, M_2)$

Input DFAs $M_1 = (Q_1, \Sigma, \delta_1, q_{01}, F_1)$ and $M_2 = (Q_2, \Sigma, \delta_2, q_{02}, F_2)$
Output DFA $M = (Q, \Sigma, \delta, q_0, F)$ with $L(M) = L(M_1) \widehat{\odot} L(M_2)$

```
1:  Q, δ, F ← ∅
2:  q0 ← [q01, q02]
3:  W ← {q0}
4:  while W ≠ ∅ do
5:      Pick [q1, q2] ∈ W
6:      Q.add([q1, q2]); W.remove([q1, q2])
7:      if (q1 ∈ F1) ⊙ (q2 ∈ F2) then
8:          F.add([q1, q2])
9:      for all a ∈ Σ do
10:         q1' ← δ1(q1, a); q2' ← δ2(q2, a)
11:         if [q1', q2'] ∉ Q then
12:             W.add([q1', q2'])
13:         δ.add([q1, q2] -a-> [q1', q2'])
```

Language operations $\widehat{\odot}$	$b_1 \odot b_2$
Union	$b_1 \vee b_2$
Intersection	$b_1 \wedge b_2$
Set difference $(L_1 \setminus L_2)$	$b_1 \wedge \neg b_2$

Nondeterministic finite automata

few comments about NFA

- When a machine is at a given state and reads an input symbol, there is precisely one choice of its next state. This is call deterministic computation.
- In nondeterministic machines, multiple choices may exist for the next state.
- A deterministic finite automaton is abbreviated as DFA; a nondeterministic finite automaton is abbreviated as NFA.
- A DFA is also an NFA.
- Since NFA allow more general computation, they can be much smaller than DFA.

NFA N_4 example

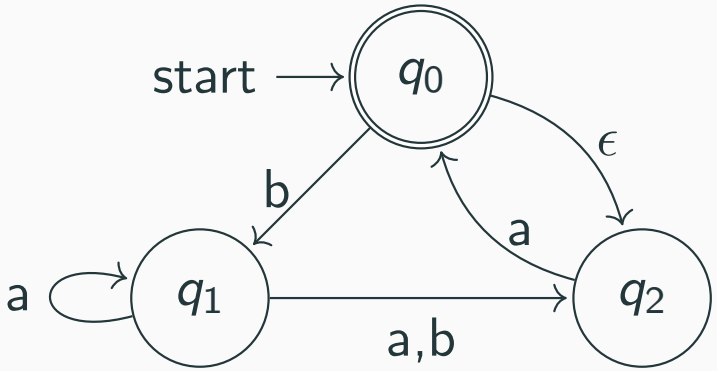


Figure 10: The NFA N_4

- on input string baa, N_4 has several possible computation:
  - $q_0 \xrightarrow{b} q_1 \xrightarrow{a} q_1 \xrightarrow{a} q_1$
  - $q_0 \xrightarrow{b} q_1 \xrightarrow{a} q_2 \xrightarrow{a} q_0$
  - $q_0 \xrightarrow{b} q_1 \xrightarrow{a} q_1 \xrightarrow{a} q_2$

Example N_5

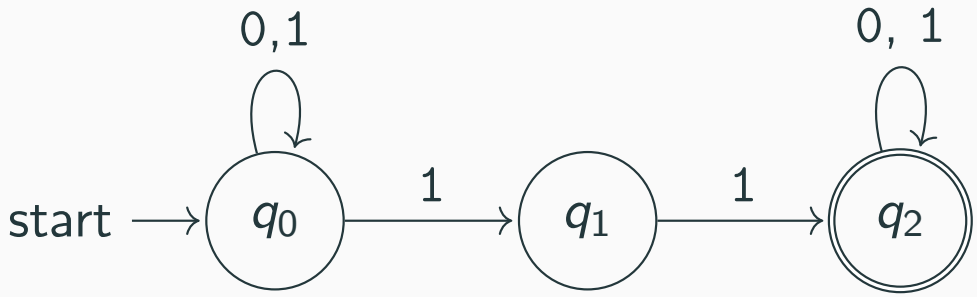


Figure 11: The NFA N_5

- on input word 10110, N_5 has
  - $q_0 \xrightarrow{1} q_1$
  - $q_0 \xrightarrow{1} q_0 \xrightarrow{0} q_0 \xrightarrow{1} q_1 \xrightarrow{1} q_2 \xrightarrow{0} q_2$.
  - ...etc

NFA – formal definition

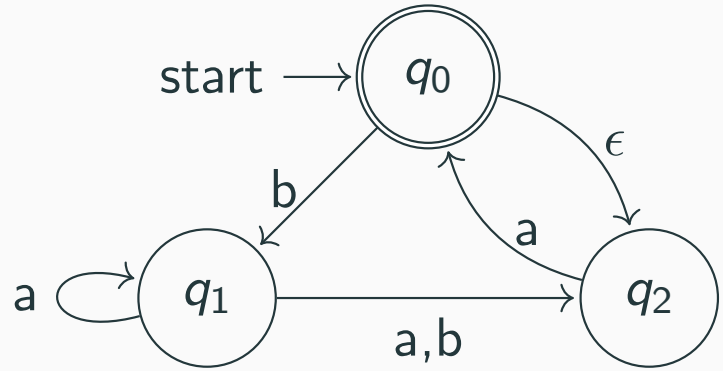
Defn. (NFA)
A nondeterministic finite automation M is a 5-tuple $(Q, \Sigma, \delta, q_0, F)$ where

- Q is a finite set of states;
- Σ is a finite alphabet;
- $\delta : Q \times \Sigma_\epsilon \to 2^Q$ is the transition function, where $\Sigma_\epsilon := \Sigma \cup \{\epsilon\}$, 2^Q is the power set of Q;
- $q_0 \in Q$ is the start state; and
- $F \subseteq Q$ is the set of accepting (or final) states.

□

note that the transition function accepts the empty string ϵ as an input symbol.

the NFA N_4 – formal definition



- $N_4 = (\{q_0, q_1, q_2\}, \{\mathtt{a}, \mathtt{b}\}, \delta, q_0, \{q_0\})$ is a NFA with transition function

$$\delta = \begin{array}{c|ccc} & \epsilon & \mathtt{a} & \mathtt{b} \\ \hline q_0 & \{q_2\} & \emptyset & \{q_1\} \\ q_1 & \emptyset & \{q_1, q_2\} & \{q_2\} \\ q_2 & \emptyset & \{q_0\} & \emptyset \end{array}.$$

formal definition of computation

Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA and $w = a_1 \cdots a_n$ be a word with $a_i \in \Sigma_\epsilon$ for all $i \in [n]$.

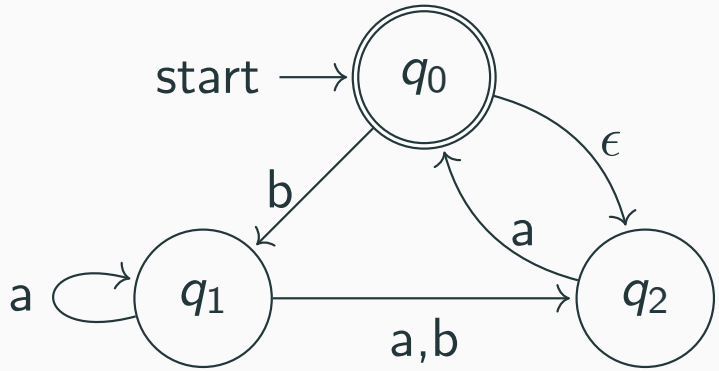
Defn.

- a run of N on w starting from p_0 is the sequence $p_0\ a_1\ p_1\ a_2\ \ldots\ a_n\ p_n$, where $p_i \in Q$ for all $i \in [n] \cup \{0\}$, $p_i \in \delta(p_{i-1}, a_i)$ for all $i \in [n]$.
- a run of N on w is defined as a run of N on w starting from q_0.
- a run of N on w is called accepting if $p_n \in F$.
- N accepts w if there exists an accepting run of N on w
- the language of N, denoted by $L(N)$ is defined to be the set of all strings accepted by N.

□

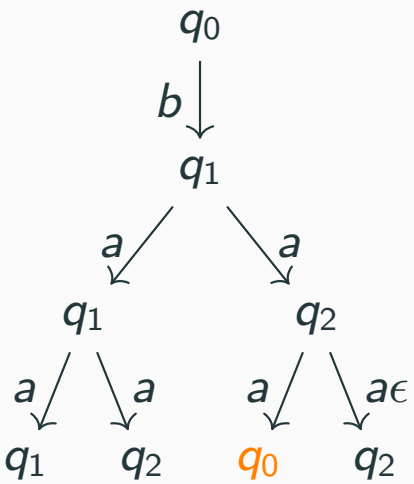
some remarks and example N_4

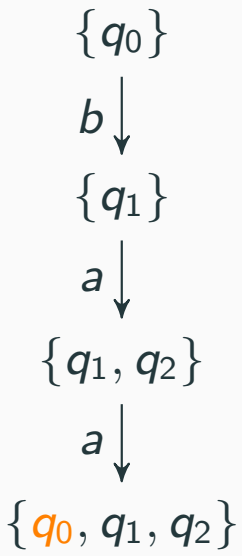
- note that finitely many empty strings can be inserted in w.
- the existence of an accepting run is sufficient to show the acceptance of an input string.



- on input string baa, there are several possible runs:
  - $q_0 \xrightarrow{b} q_1 \xrightarrow{a} q_1 \xrightarrow{a} q_1$
  - $q_0 \xrightarrow{b} q_1 \xrightarrow{a} q_2 \xrightarrow{a} q_0$ $\leftarrow$ this is accepting
  - $q_0 \xrightarrow{b} q_1 \xrightarrow{a} q_1 \xrightarrow{a} q_2$

Thm. (equivalence between NFA/DFA)
Every NFA has an equivalent DFA. That is, for every NFA N, there is a DFA M such that $L(M) = L(N)$. □





Proof.
Let $N = (Q, \Sigma, \delta, q_0, F)$ be an NFA. For $R \subseteq Q$, define

$$E(R) := \{q \in Q \mid q \text{ can be reached from } R \text{ along 0 or more } \epsilon \text{ transitions}\}$$

Construct a DFA $M = (Q', \Sigma, \delta', q_0', F')$ with:

- $Q' := 2^Q$;
- $\delta'(R, a) := \{q \in Q \mid q \in E(\delta(r, a)) \text{ for some } r \in R\}$;
- $q_0' := E(\{q_0\})$; and
- $F' := \{R \in Q' \mid R \cap F \neq \emptyset\}$.

■

Intuitive idea: subset construction

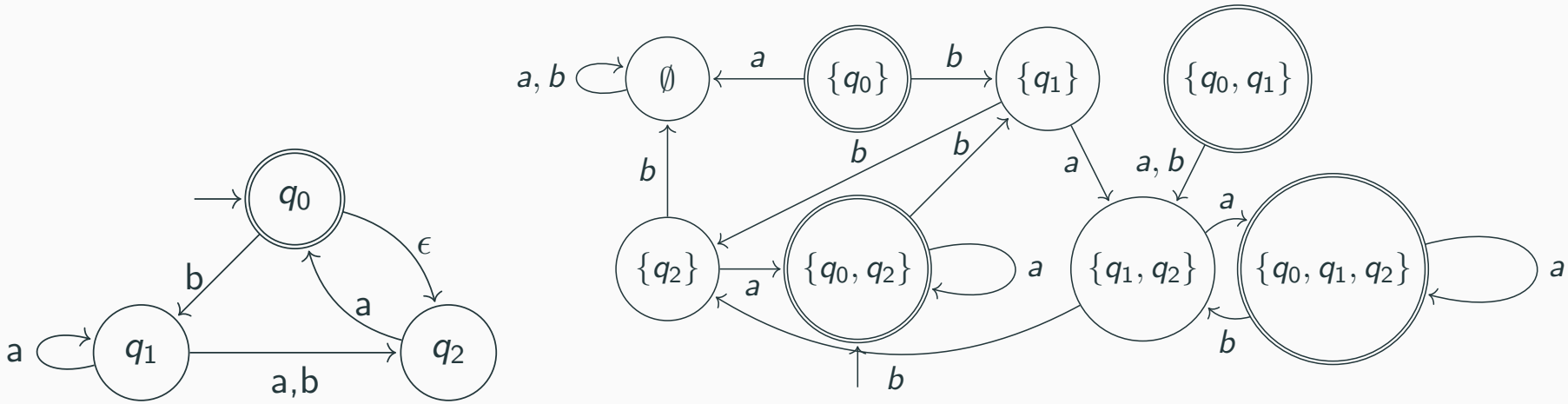


Figure 12: An equivalent DFA to N_4

Algorithm 3 Conversion NFAtoDFA(N)

Input NFA $N = (Q, \Sigma, \delta, q_0, F)$
Output DFA $M = (Q', \Sigma, \delta', q_0', F')$ with $L(M') = L(N)$

1: $Q', \delta', F' \leftarrow \emptyset$
2: Compute $E(\{q_0\})$; $q_0' \leftarrow E(\{q_0\})$; $W \leftarrow \{q_0'\}$
3: **while** $W \neq \emptyset$ **do**
4: pick $q' \in W$; W.remove(q')
5: compute $E(\{q'\})$; Q'.add($E(\{q'\})$)
6: **if** $E(\{q'\}) \cap F \neq \emptyset$ **then**
7: F'.add($E(\{q'\})$)
8: **for all** $a \in \Sigma$ **do**
9: $q'' \leftarrow E(\bigcup_{q \in q'} \delta(q, a))$
10: **if** $q'' \notin Q'$ **then**
11: W.add(q'')
12: δ'.add($q' \xrightarrow{a} q''$)

remarks

- the construction of DFA in the proof is called the subset construction.
- succinctness of NFA: from the construction, we see that the number of states of the DFA is exponentially growth in the number of states of the equivalent NFA in worst case.

eg.

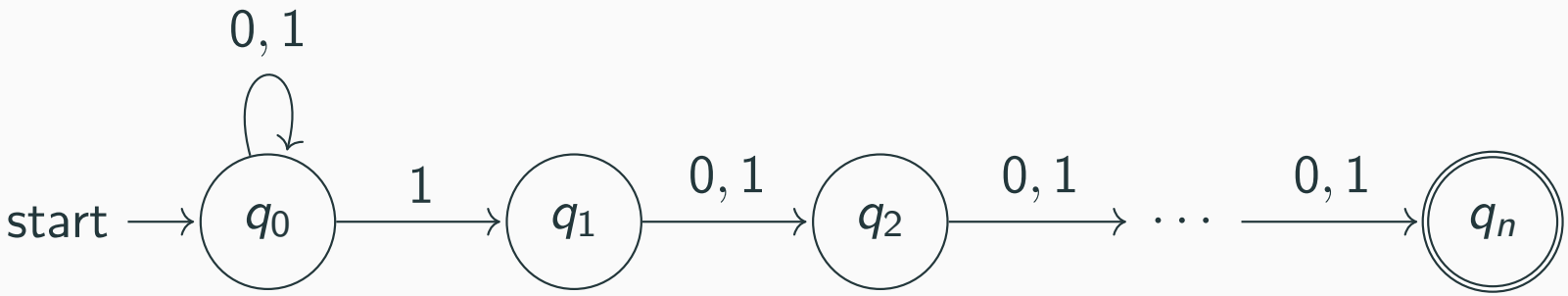
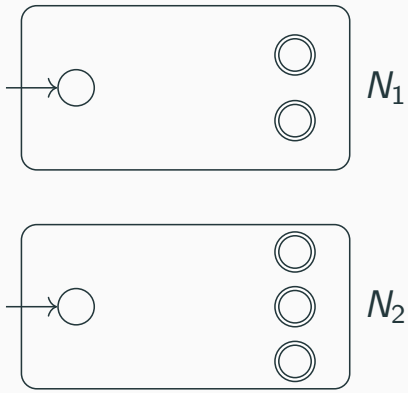


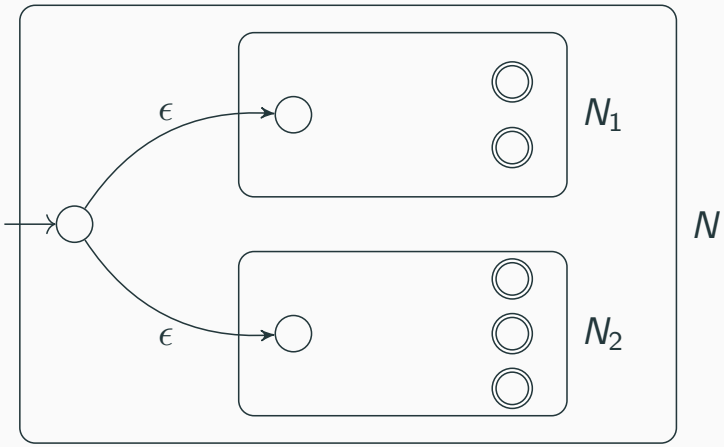
Figure 13: this NFA has no equivalent DFA with fewer than 2^n states

closure properties – revisited

Thm.
The class of regular languages is closed under the union operation. □



(a) NFA N_1 and N_2



(b) NFA N

Proof.
Let $N_i = (Q_1, \Sigma, \delta_i, p_i, F_i)$, $i = 1, 2$ be two NFA s. Construct $N = (Q, \Sigma, \delta, q_0, F)$ as

- $Q := Q_1 \cup Q_2 \cup \{q_0\}$;
- $F := F_1 \cup F_2$;
- $\delta(q, a) =$

$$\begin{cases} \delta_1(q, a) & q \in Q_1 \\ \delta_2(q, a) & q \in Q_2 \\ \{p_1, p_2\} & q = q_0 \wedge a = \epsilon \\ \emptyset & q = q_0 \wedge a \neq \epsilon \end{cases}.$$

■

Algorithm for NFA union

Algorithm 4 NFA union UnionNFA(N_1, N_2)

Input NFA $N_1 = (Q_1, \Sigma, \delta_1, p_1, F_1)$ and $N_2 = (Q_2, \Sigma, \delta_2, p_2, F_2)$
Output NFA $N = (Q, \Sigma, \delta, q_0, F)$ with $L(N) = L(N_1) \cup L(N_2)$

1: $Q \leftarrow Q_1 \cup Q_2 \cup \{q\}$ for a fresh q; $F \leftarrow F_1 \cup F_2$; $q_0 \leftarrow q$
2: $\delta \leftarrow \delta_1 \cup \delta_2 \cup \{q_0 \xrightarrow{\epsilon} p_1, q_0 \xrightarrow{\epsilon} p_2\}$

remarks on other NFA boolean operations

- no direct algorithm to construct the complement from an NFA N.
- however, one can obtain the complement as $\overline{N} \leftarrow \text{CompDFA}(\text{NFAtoDFA}(N))$.
- on NFA, it is no longer possible to uniformly implement all binary boolean operations, like DFA.
- for $\text{InterNFA}(N_1, N_2)$, the production construction as in Algorithm 2 still works.
- Algorithm 2 holds for union of NFA N_1 and N_2 if both $L(N_i) \neq \emptyset, i = 1, 2$.
- however, Algorithm 4 is much more efficient.
- Algorithm 2 fails to hold for set difference of NFA.
- however, the set difference of NFA can be constructed from $L(N_1) \cap \overline{L(N_2)}$.

closure properties under regular operations

Defn. (regular op.)
Let A and B be languages over Σ. The regular operations are Boolean operations and

- concatenation: $A \circ B := \{xy \mid x \in A \wedge y \in B\}$.
- (Kleene) Star: $A^* := \{x_1 x_2 \dots x_k \mid k \geq 0 \wedge \forall i \in [k].x_i \in A\} = \bigcup_{n \in \mathbb{N}_0} A^{\circ n}$.

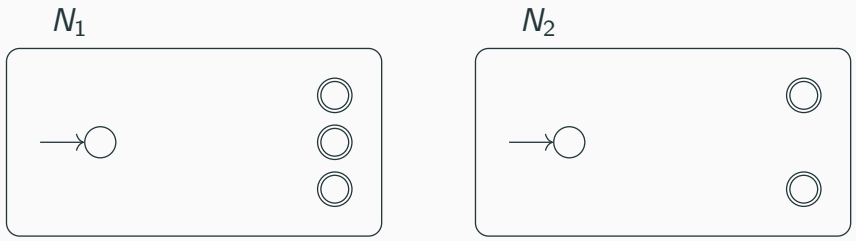
□

Thm. (closure under regular operation)
The class of regular languages is closed under regular operations. □

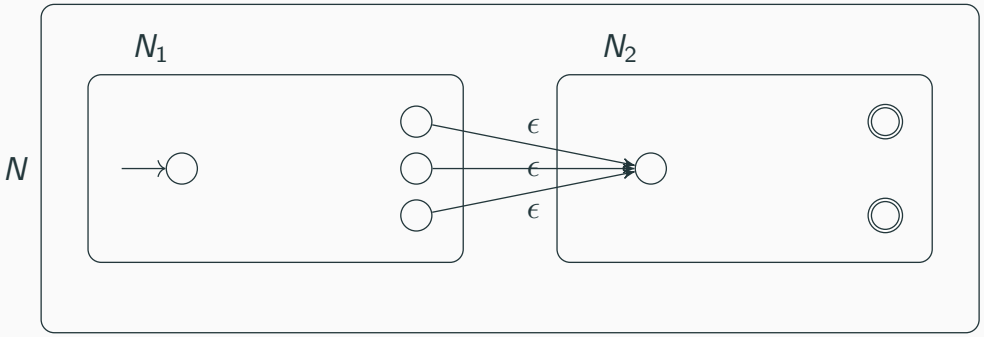
closure properties under concatenation

Thm.
The class of regular languages is closed under concatenation operation. □

Proof.



(a) NFA N_1 and N_2



(b) NFA N

Let $N_i = (Q_i, \Sigma, \delta_i, p_i, F_i)$ recognize L_i for $i = 1, 2$. Construct $N = (Q, \Sigma, \delta, p_1, F_2)$ as:

- $Q := Q_1 \cup Q_2$;
- $\delta(q, a) :=$

$$\begin{cases} \delta_1(q, a) & q \in Q_1 \wedge q \notin F_1 \\ \delta_1(q, a) & q \in F_1 \wedge a \neq \epsilon \\ \delta_1(q, a) \cup \{p_2\} & q \in F_1 \wedge a \neq \epsilon \\ \delta_2(q, a) & q \in Q_2 \end{cases}$$

■

algorithm for NFA concatenation

Algorithm 5 NFA concatenation ConcatNFA(N_1, N_2)

Input NFA $N_1 = (Q_1, \Sigma, \delta_1, p_1, F_1)$ and $N_2 = (Q_2, \Sigma, \delta_2, p_2, F_2)$

Output An NFA $N = (Q, \Sigma, \delta, q_0, F)$ with $L(N) = L(N_1) \circ L(N_2)$

1: $Q \leftarrow Q_1 \cup Q_2$; $F \leftarrow F_2$; $q_0 \leftarrow p_1$;

2: $\delta \leftarrow \delta_1 \cup \delta_2$

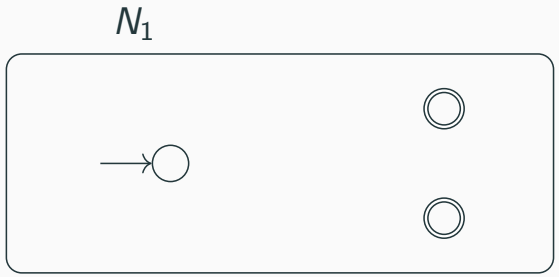
3: **for all** $q \in F_1$ **do**

4: $\quad \delta.\text{add}(q \xrightarrow{\epsilon} p_2)$

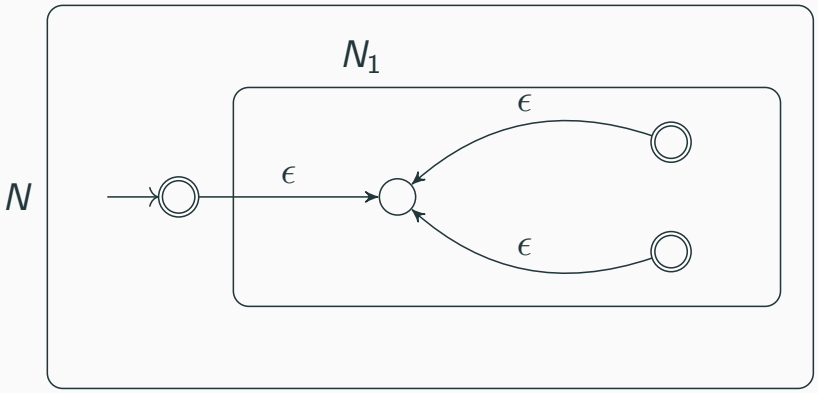
closure properties under Kleene star

Thm.
The class of regular language is closed under star operation. □

Proof.



(a) NFA N_1



(b) NFA N

Let $N_1 = (Q_1, \Sigma, \delta_1, p_1, F_1)$ recognizes L_1. Construct $N = (Q, \Sigma, \delta, q_0, F)$ where

- $Q := \{q_0\} \cup Q_1$;
- $F := \{q_0\} \cup F_1$;
- $\delta :=$

$$\begin{cases} \delta_1(q,a) & q \in Q_1 \wedge q \notin F_1 \\ \delta_1(q,a) & q \in F_1 \wedge a \neq \epsilon \\ \delta_1(q,a) \cup \{p_1\} & q \in F_1 \wedge a = \epsilon \\ \{p_1\} & q = q_0 \wedge a = \epsilon \\ \emptyset & q = q_0 \wedge a \neq \epsilon \end{cases}.$$

algorithm for NFA star

Algorithm 6 NFA star StarNFA(N)

Input NFA $N = (Q, \Sigma, \delta, q_0, F)$

Output An NFA $N' = (Q', \Sigma, \delta', q_0', F')$ with $L(N') = L(N)^*$

1: $Q'.\text{add} \leftarrow Q \cup \{q'\}$ for a fresh q'; $q_0' \leftarrow q'$; $F' \leftarrow F \cup \{q_0'\}$

2: $\delta' \leftarrow \delta \cup \{q_0' \xrightarrow{\epsilon} q_0\}$

3: **for all** $q \in F$ **do**

4: $\quad \delta'.\text{add}(q \xrightarrow{\epsilon} q_0)$

Regular expression

Examples of regular expressions

Regular expressions serve as the input language for many systems that process strings.

- Search commands such as `UNIX grep` or equivalent commands for finding strings that one see in the Web browsers or text-formatting systems. These systems use a regular-expression-like notation fro describing patterns that the user wants to find in a file. Different search systems convert the regular expression into either a DFA or an NFA and simulate that automation on the file being searched.
- Lexical-analyzer generators, such as Lex or Flex. A lexical analyzer is the component of a compiler that breaks the source program into logical units (*token*) of one or more characters that have a shared significance. Examples of tokens include keywords (e.g. `while`), identifier (e.g. any letter followed by zero or more letters or digits), and signs such as $<=$ or $+$. A lexical-analyzer generator accepts descriptions of the forms of tokens, which essentially are regular expressions, and produces DFA that recognizes which token appears on the input.

the syntax of regular expression

Let Σ be an alphabet.

Defn. (regular expression)
A regular expression r over Σ is a finite production of the following (regular) grammar:

$$
\begin{array}{rcll}
r & ::= & \emptyset & \text{empty set} \\
 & | & \epsilon & \text{empty expression} \\
 & | & a & \forall a \in \Sigma \text{ symbols} \\
 & | & r^* & \text{(Kleene) star} \\
 & | & r \mid r & \text{alternating/choice} \\
 & | & r\ r & \text{concatenation.}
\end{array}
$$

□

regular expression is usually abbreviated as `regex`.

Remark.

- we sometimes write $r^+ := rr^*$, hence $r^* = r^+|\epsilon$.
- we write $r^k := \underbrace{rr \cdots r}_{k \text{ times}}$ and define $r^0 := \epsilon$.
- sometimes $r_1 \mid r_2$ is written as $r_1 + r_2$.

□

Example

Let $\Sigma = \{a, b\}$.

- $\emptyset$ is a regular expression, so are $\emptyset\emptyset$ and $\emptyset a$ where $a \in \Sigma$.
- $(ab)a$ is a regular expression and is usually written as aba.
- $(ab)^*$ is a regular expression, so are $((ab)^*)^*$, ab^*, $(ab)^* \mid ab^*$.

languages described by a regular expression

Defn. (semantics of regular expression)
A regular expression r over Σ defines the language $L(r)$ over the same alphabet Σ as follows:

- If $r = \emptyset$, then $L(\emptyset) = \emptyset$.
- If $r = a$ for some $a \in \Sigma$ or $a = \epsilon$, then $L(r) = \{a\}$.
- If r is of the form $r_1 r_2$, then $L(r) = L(r_1) \circ L(r_2)$.
- If r is of the form $r_1 \mid r_2$, then $L(r) = L(r_1) \cup L(r_2)$.
- If r is of the form $(r_1)^*$, then $L(r) = (L(r_1))^*$.

□

Remark.

- $\emptyset \neq \{\epsilon\}$ and $\emptyset^k = \emptyset$ for all $k \in \mathbb{N}$. however, $\emptyset^* = \{\epsilon\}$, $\emptyset^0 = \{\epsilon\}$.

□

examples

- for $a, b \in \Sigma$, $L(ab) = L(a)L(b) = \{a\}\{b\} = \{ab\}$.
- $L((a|b)^*) = (L(a|b))^* = (\{a\} \cup \{b\})^* = \{a, b\}^*$.
- $L(0^*10^*) = \{w \mid w \text{ contains a single } 1\}$.
- $L(\Sigma^*1\Sigma^*) = \{w \mid w \text{ has at least a } 1\}$.
- $L((\Sigma\Sigma)^*) = \{w \mid w \text{ is a string with even length}\}$.
- $L((0|\epsilon)(1|\epsilon)) = \{\epsilon, 0, 1, 01\}$.
- $L(1^*\emptyset) = \emptyset$. $L(r\emptyset) = L(r)L(\emptyset) = L(r)\emptyset = \emptyset$.
- $L(\emptyset^*) = \{\epsilon\}$.
- for any regular expression r, we have $L(r|\emptyset) = L(r)$ and $L(r\epsilon) = L(r)$.

Thm. (regular expression is regular language)
Regular expressions define precisely the class of regular languages. More precisely,

- for every regular expression r over Σ, $L(r)$ is a regular language, i.e., there is an NFA N such that $L(N) = L(r)$;
- for every NFA N, there is a regular expression r such that $L(r) = L(N)$.

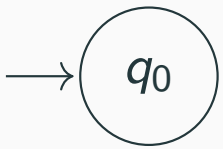
□

first part of the Theorem

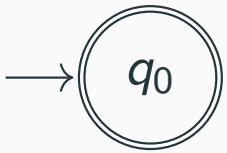
Thm.
For every regular expression r over Σ, $L(r)$ is a regular language, i.e., there is an NFA N such that $L(N) = L(r)$. □

Proof. By induction on the `regex` r.

- $r = \emptyset$. Then $L(r) = \emptyset$. Construct the NFA $N_\emptyset = (\{q_0\}, \Sigma, \delta, q_0, \emptyset)$:



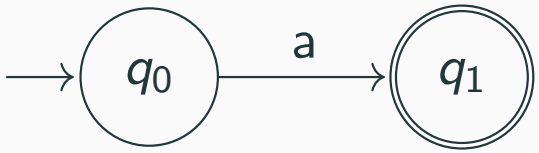
- $r = \epsilon$. Then $L(r) = \{\epsilon\}$. Construct the NFA $N_\epsilon = (\{q_0\}, \Sigma, \delta, q_0, \{q_0\})$:



first part of the Theorem

Proof (Cont.)

- $r = a$ for $a \in \Sigma$. Then $L(r) = \{a\}$. Construct the NFA $N_a = (\{q_0, q_1\}, \Sigma, \delta, q_0, \{q_1\})$:

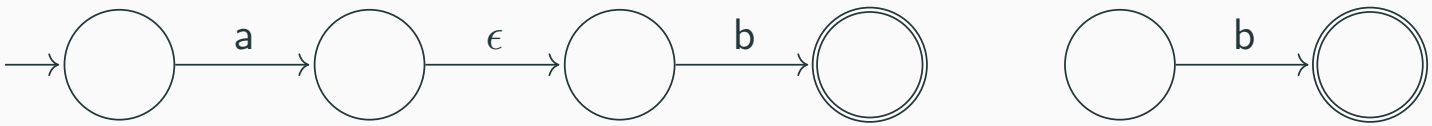


- for $r = r_1 \mid r_2, r_1 \circ r_2$ or $r = r_1^*$, by (structural) induction hypothesis and closure properties of NFA. ■

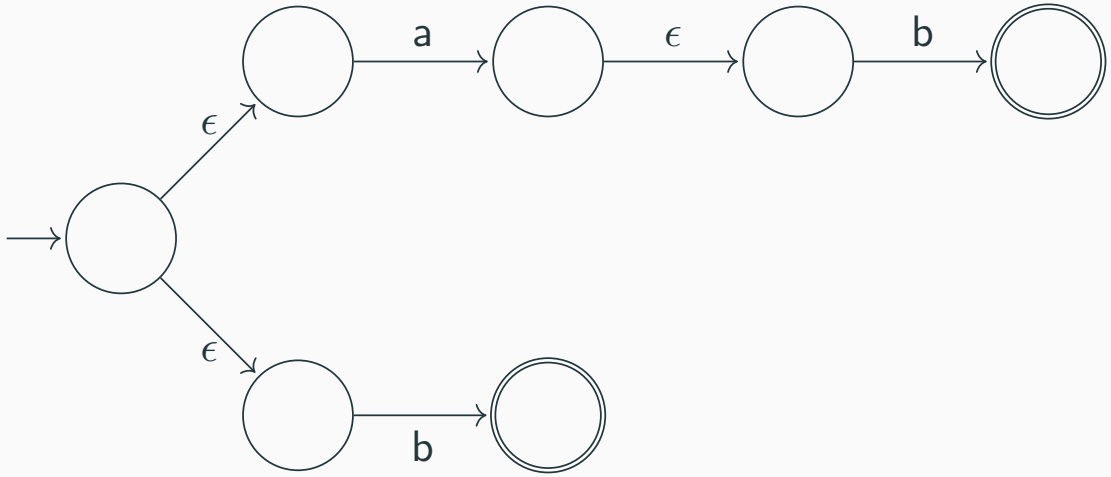
example

Example

- ab and b



- $(ab \mid b)$



second part of the Theorem

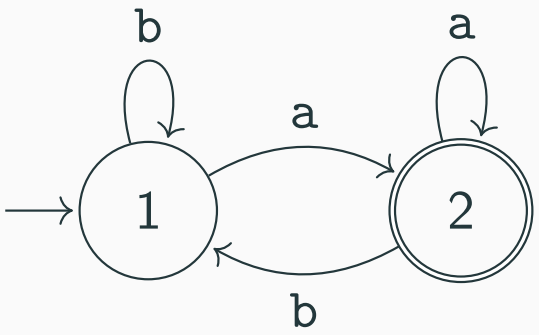
Thm.
For every NFA N there is a regular expression r such that $L(r) = L(N)$. □

Proof
Let $M = (Q, \Sigma, \delta, q_0, F)$ be an equivalent DFA. W.L.O.G. let $Q = \{1, \ldots, n\}$. For every $1 \leq i, j \leq n$ and $0 \leq k \leq n$, define the language $L(i, j, k)$

$$L(i, j, k) := \{w \in \Sigma^* \mid \text{there is a run of } M \text{ on } w \text{ from } i \text{ to } j \text{ without passing through any states } \geq k + 1\}.$$

That is, if $w \in L(i, j, k)$ there is a run of M on w from i to j without passing through states $k + 1, \ldots, n$.

example



	$k = 0$	$k = 1$	$k = 2$
$L(1, 1, k)$	$\mathtt{b} \mid \epsilon$	$\mathtt{b}^*$	
$L(1, 2, k)$	$\mathtt{a}$	$\mathtt{b}^*\mathtt{a}$	$(\mathtt{a} \mid \mathtt{b})^*\mathtt{a}$
$L(2, 1, k)$	$\mathtt{b}$	$\mathtt{b}^+$	
$L(2, 2, k)$	$\mathtt{a} \mid \epsilon$	$\mathtt{b}^*\mathtt{a} \mid \epsilon$	

second part of the Theorem (cont.)

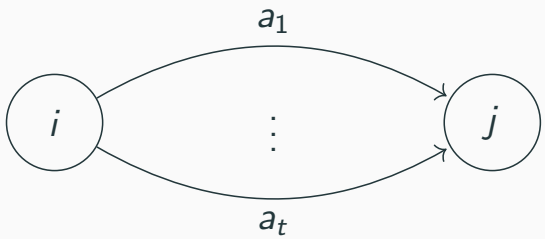
Claim (1)
For every $1 \leq i, j \leq n$ and $0 \leq k \leq n$, there exists a `regex` r such that $L(r) = L(i, j, k)$. □

Proof of Claim 1
We will prove the Claim by induction on k.

(Base case $k = 0$) for every $1 \leq i, j \leq n$, consider $L(i, j, 0)$.:

- if $i \neq j$ and there is no transition from i to j: 

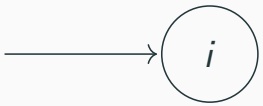
the language $L(i, j, 0) = \emptyset$, so we may take $r = \emptyset$.
- if $i \neq j$ and there are some transition from i to j:



then $L(i, j, 0) = \{a_1, \ldots a_t\}$, so we may take $r = a_1 \mid \cdots \mid a_t$.

proof of the Theorem (cont.)

- if $i = j$ and there is no transition from i to i:
  then $L(i, i, 0) = \{\epsilon\}$ and we may take $r = \epsilon$.



- if $i = j$ and there are some transitions from i to i:

  then $L(i, i, 0) = \{a_1, \ldots, a_t, \epsilon\}$, we can take $r = a_i \mid \cdots \mid a_t \mid \emptyset^*$.

(Induction step) We have the following identity:

$$L(i, j, k+1) = L(i, j, k) \cup \big(L(i, k+1, k) \circ L(k+1, k+1, k)^* \circ L(k+1, j, k)\big).$$

By induction hypothesis, there is `regex` for each of $L(i, j, k)$, $L(i, k+1, k)$, $L(k+1, k+1, k)$ and $L(k+1, j, k)$. Thus there is `regex` for $L(i, j, k+1)$. ■

proof of the Theorem (cont.)

Finishing the proof of Theorem
By definition of $L(M)$, we have

$$L(M) = \bigcup_{q_f \in F} L(q_0, q_f, n).$$

By Claim 1, there is a `regex` for each $L(q_0, q_f, n)$.

By taking the union over all $q_f \in F$, we get a `regex` for $L(M)$. ■

Conclusion

Thm.
Let L be a language. The following are equivalent:

- L is accepted by a DFA, i.e., L is a regular language.
- L is accepted by an NFA.
- L is defined by a regular expression.

□

Remark.

- one nice implication of this corollary is that languages defined by regular expression are also closed under intersection and complement.
- this is despite the fact that we are not allowed to use intersection or negation in regular expressions.
  - in practice, there is an extended version of `regex` that allows using intersection and negation (c.f. [VVE25])

□

another approaches of `regex` to NFA

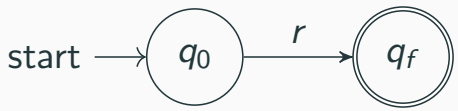
One can generalize the definition of NFA to the one allowing transitions labeled by regular expressions. Denote by $\mathcal{RE}(\Sigma)$ the set of all regular expressions over an alphabet Σ.

Defn. (NFA-reg)
A nondeterministic automation with regular expression transitions (NFA-reg) is a tuple $N = (Q, \Sigma, \delta, q_0, F)$ where

- Q, Σ, q_0, F are as for NFA;
- $\delta : Q \times \mathcal{RE}(\Sigma) \to 2^Q$ such that $\delta(q, r) = \emptyset$ for all but a finite number of pairs $(q, r) \in Q \times \mathcal{RE}(\Sigma)$.

Accepting runs are defined as for NFA. An NFA-reg A accepts a word $w \in \Sigma^*$ if A has an accepting run on $r_1 \cdots r_k$ such that $w \in L(r_1) \cdots L(r_k)$. □

Then, given an `regex` r, one can construct an NFA-reg as



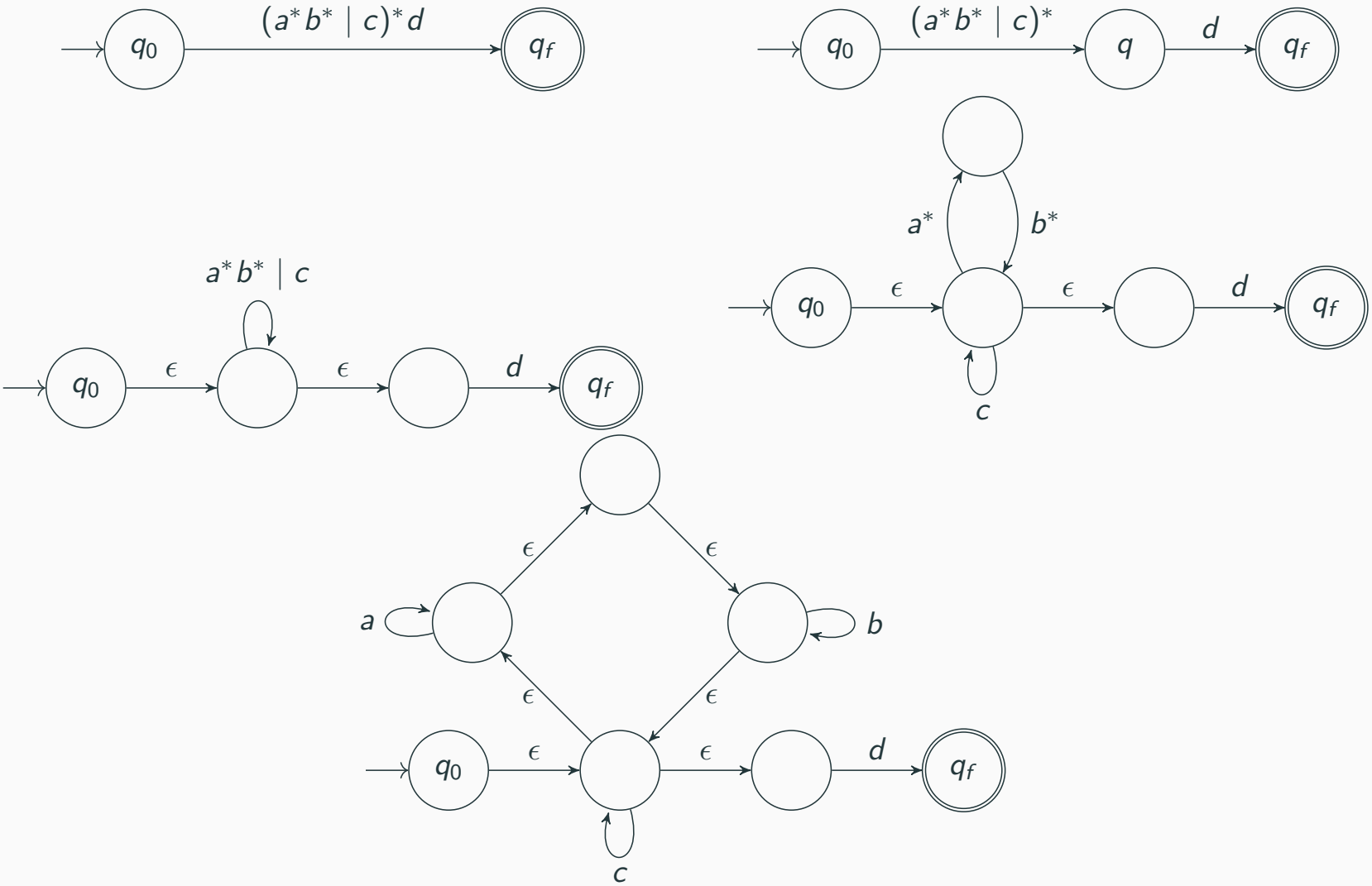
rules for NFA-reg to NFA

- Concatenation:

- Choice

- Kleene star:

NFA-reg to NFA example



another approaches of NFA to `regex`

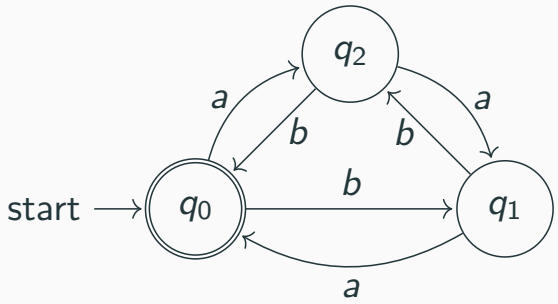
The proof we have seen is called the transitive closure method [HMU07]. However there are two other famous methods:

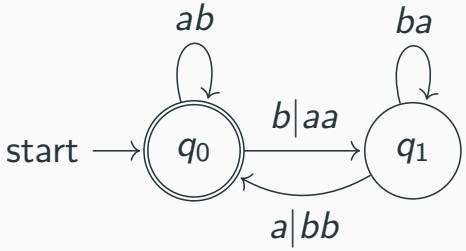
- State removal method [EB23]:
  1. Unify all final states into a single final state using ϵ-transitions.
  2. Unify all multi-transitions into a single transition that contains union of inputs.
  3. Remove states and change the transitions accordingly until there is only the initial state and the final state.
  4. Compute the regular expression.
- Brzozowski algebraic method [Brz64]:
  1. Construct an equivalent NFA $N = (Q, \Sigma, \delta, q_0, F)$ without ϵ-transitions.
  2. For every q_i construct the equation

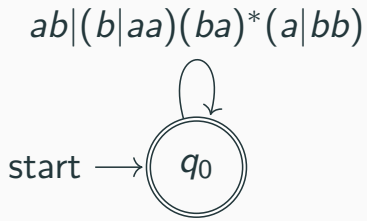
$$Q_i = \bigcup_{q_i \xrightarrow{a} q_j} aQ_j \cup \begin{cases} \{\epsilon\} & \text{if } q_i \in F \\ \emptyset & \text{else} \end{cases}$$

     .
  3. Solve the equation and find Q_0.

example of state removal method

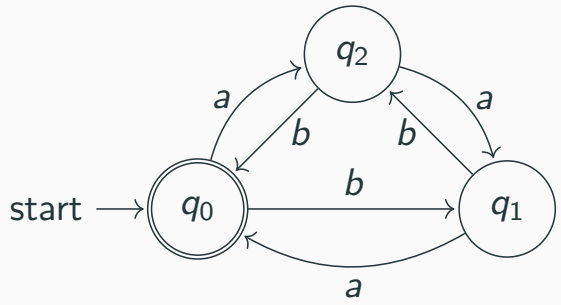






We get the `regex` $R = \big(ab|(b|aa)(ba)^*(a|bb)\big)^*$.

example of Brzozowski's algebraic method



$$Q_0 = bQ_1 + aQ_2 + \epsilon$$
$$Q_1 = aQ_0 + bQ_2$$
$$Q_2 = bQ_0 + aQ_1 = bQ_0 + a(aQ_0 + bQ_2) = abQ_2 + (b + aa)Q_0$$

By Arden's Lemma: $L = UL \cup V$ iff $L = U^*V$ where $\epsilon \notin U$, we get $Q_2 = (ab)^*(b + aa)Q_0$.

$$Q_0 = b(aQ_0 + bQ_2) + aQ_2 + \epsilon = (ba + (bb + a)(ab)^*(b + aa))Q_0 + \epsilon.$$

By Arden's Lemma again, we have $Q_0 = (ba + (bb + a)(ab)^*(b + aa))^*$.

More closure properties of regular languages

closure under difference

Thm. (closure under difference)
If L and M are two regular languages, then so is $L \setminus M$. □

Proof.

- observe that $L \setminus M = L \cap \overline{M}$.
- M regular $\implies$ $\overline{M}$ by closure property under complement.
- hence $L \cap \overline{M}$ is regular by closure under intersection.

■

reversal

Defn. (reversal)
The reversal of a string $w = a_1 a_2 \cdots a_n$ is $w^R := a_n a_{n-1} \cdots a_2 a_1$.
The reversal of a language L is defined to be $L^R := \{w^R \mid w \in L\}$. □

e.g. if $L = \{001, 10, 111\}$, then $L^R = \{100, 01, 111\}$.

Thm. (closure under reversal)
If L is a regular language, so is L^R. □

Proof.

- if L is recognized by an NFA N, reverse all the transitions in the diagram of N.
- in the new automation N^R, let $F^R = \{q_0\}$.
- create a new initial state p_0 with ϵ-transitions to all accepting states in F.
- if L is represented by a `regex` E, either convert it into an NFA or prove directly by induction.

■

homomorphism

Defn. (homomorphism)
A string homomorphism is a function on strings that works by substituting a particular string from each symbol. More precisely, if h is a homomorphism on Σ and $w = a_1 \cdots a_n \in \Sigma^*$, then $h(w) := h(a_1)h(a_2)\cdots h(a_n)$. □

- e.g. $h(0) = ab$, $h(1) = \epsilon$ is a string homomorphism. $h(0011) = abab$.
- extended to language: $h(L) := \{h(w) \mid w \in L\}$.

Thm. (close under homomorphism)
If L is a regular language over Σ and h is a homomorphism on Σ, then $h(L)$ is regular. □

closure under homomorphism

Proof.
Let $L = L(R)$ for some `regex` R. Let $h(R)$ be the `regex` by replacing each symbol a of Σ in R by $h(a)$. Claim: $h(L) = L(h(R))$. We will proceed by induction:

- Base: if $R = \epsilon$ or $\emptyset$, then $h(R) = \epsilon$ or $\emptyset$ respectively. $h(L(R)) = L(h(R))$ in either case.
- if $R = a$ for some $a \in \Sigma$. $L(R) = \{a\}$ and $h(L(R)) = \{h(a)\}$. However $h(R) = h(a)$ and hence $L(h(R)) = \{h(a)\}$. Thus $h(L(R)) = L(h(R))$.
- Inductive: $R = R_1 \mid R_2$ we have $h(R) = h(R_1) \cup h(R_2)$. so
  1. $L(h(R)) = L(h(R_1) \mid h(R_2)) = L(h(R_1)) \cup L(h(R_2))$.
  2. $h(L(R)) = h(L(R_1 \mid R_2)) = h(L(R_1)) \cup h(L(R_2))$.

  thus $h(L(R)) = L(h(R))$ holds by induction.
- the cases $R = (R_1)^*$ and $R = R_1 R_2$ are similar.

inverse homomorphism

Thm. (inverse homomorphism)
If h is a homomorphism from Σ to alphabet $\mathcal{T}$ and L is a regular language over $\mathcal{T}$, then $h^{-1}(L)$ is also regular. □

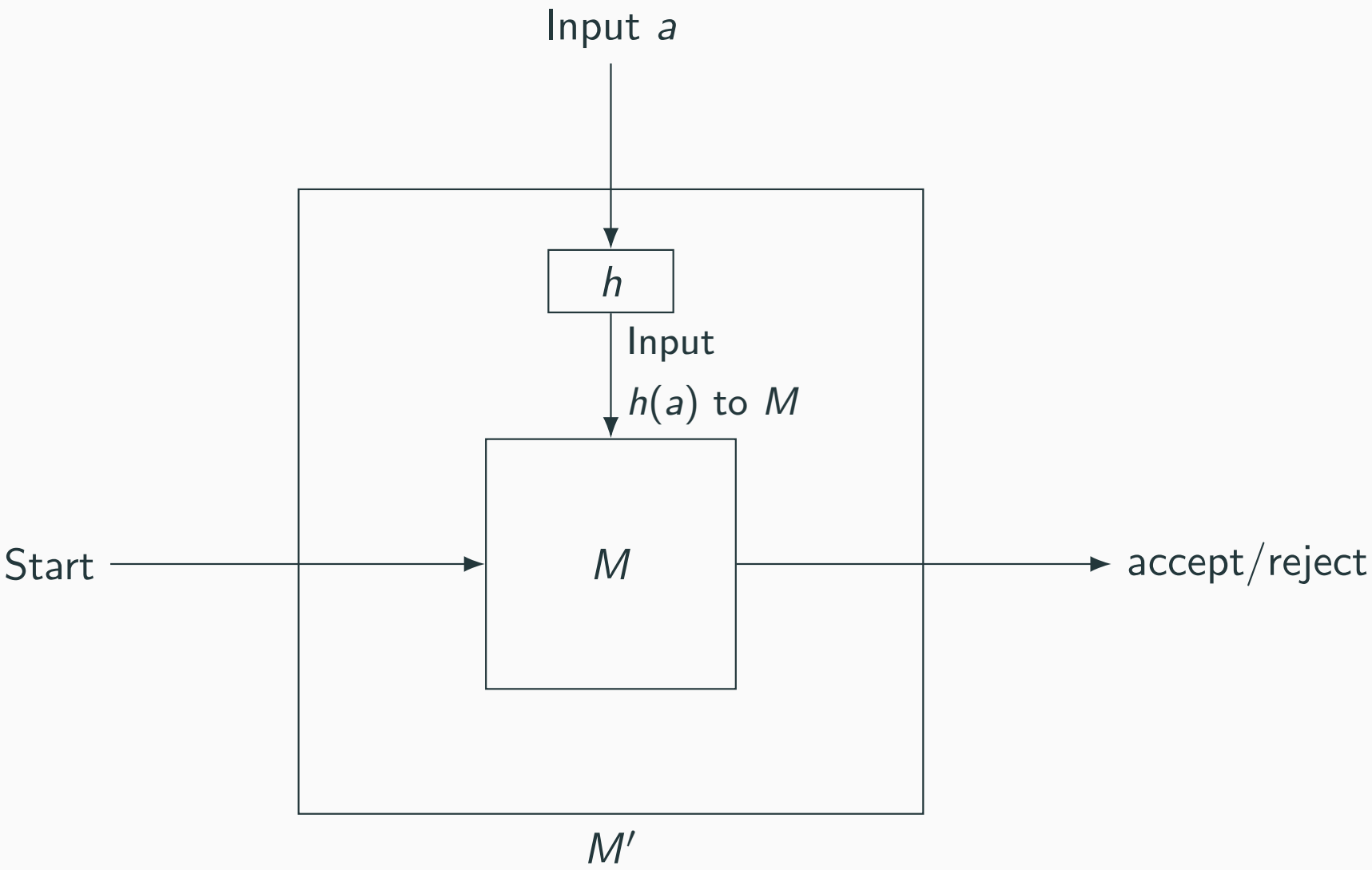
Intuition: construct a DFA for $h^{-1}(L)$ by applying h to its input and simulating the DFA for L.

Proof.

- let $M = (Q, \mathcal{T}, \delta, q_0, F)$ be a DFA s.t. $L(M) = L$.
- construct a DFA $M' = (Q, \Sigma, \gamma, q_0, F)$ by setting $\gamma(q, a) := \hat{\delta}(q, h(a))$.
- easy induction on $|w|$ to show that $\hat{\gamma}(q_0, w) = \hat{\delta}(q_0, h(w))$.

(here, $\hat{\delta}(q, b_1 \cdots b_n)$ is defined inductively as $\delta(\hat{\delta}(q, b_1 \cdots b_{n-1}), b_n)$ and $\hat{\delta}(q, b) = \delta(q, b)$ if $b \in \mathcal{T}$ and similarly for $\hat{\gamma}$) ■

inverse homomorphism



Decision properties of regular languages

fundamental questions about languages

- Emptiness: does a language describe empty?
- Universality: does a language describe Σ^*?
- Membership: is a particular string w in the described language?
- Equivalence: do two descriptions of a language actually describe the same language?

emptiness

Task: given a regular language L, decide whether $L = \emptyset$.

- If the language L is represented by an automation M, i.e., $L = L(M)$. The emptiness problem is equivalent to the graph-reachability from the initial state to an accepting state.
  - Base case: the initial state is surely reachable.
  - Induction: if state q is reachable from the initial state and there is a transition from q to p, then p is reachable.

  If M has n states, the reachability calculation takes no more time than $O(n^2)$.
- If L is represented by a `regex` r. We can, either convert r into an NFA N (within $O(n)$ time) and apply the above algorithm; or consider the following algorithm:
  - Base case: $L(\emptyset) = \emptyset$; $L(\epsilon)$ and $L(a)$ for any $a \in \Sigma$ are not.
  - Induction:
    1. $r = r_1 \mid r_2$: $L(r) = \emptyset$ iff $L(r_1) = L(r_2) = \emptyset$.
    2. $r = r_1 r_2$: $L(r) = \emptyset$ iff $L(r_1) = \emptyset \vee L(r_2) = \emptyset$.
    3. $r = r_1^*$: $L(r) \neq \emptyset$ since $L(r_1^*)$ always contains ϵ.

membership
Task: given L and $w \in \Sigma^*$, decide whether $w \in L$.

- If L is representation by an automation M, then we just need to feed w into M by simulating M on input w and get the answer.
- If M is DFA, the complexity is $O(|w|)$.
- If M is NFA, the complexity is $O(|w|s^2)$ where $s = |Q_M|$.
- If L is represented by a `regex` r of size s_r, convert it into an NFA N (with $|Q_N| = 2s_r$) in $O(s_r)$ time and perform the above simulation taking $O(|w|s_r^2)$ time.

universality
Task: given L, decide whether $L = \Sigma^*$.

- If L is represented by a DFA M, then it can be done in at most $O(n^2)$ time
- If L is represented by an NFA N, then it is **PSPACE**-complete [EB23, Sec. 3.2.6]
- If L is represented by a `regex`, then it is **PSPACE**-complete [AHU74, Thm 10.14]

Equivalence and minimization

equivalence of states

Defn. (equivalence of states)
Given a DFA $M = (Q, \Sigma, \delta, q_0, F)$ and $p, q \in Q$.

- p and q are said to be equivalent iff for all input strings $w \in \Sigma^*$,

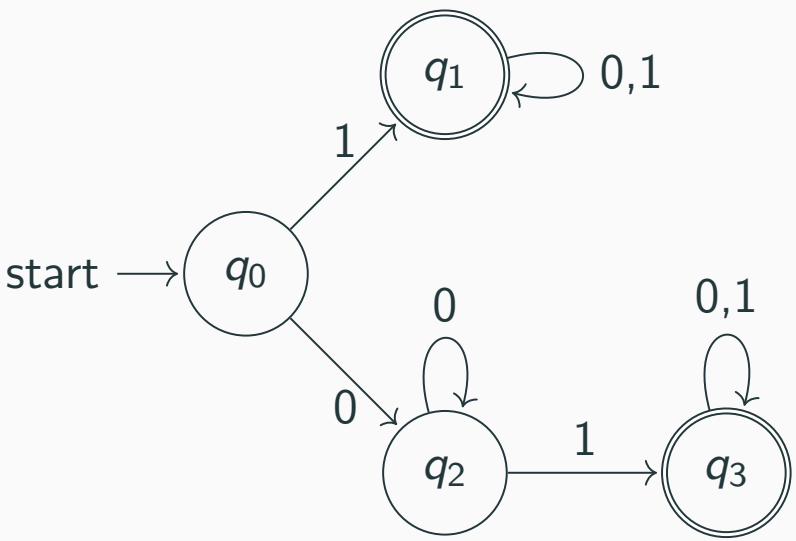
$$p \xrightarrow{w} p' \in F \iff q \xrightarrow{w} q' \in F$$

- p and q are distinguishable iff they are not equivalent.

□

note: p' need not to be q'; we only ask if both p' and q' are in F.

table-filling algorithm



- consider the DFA on the left
- since $q_0 \notin F$ but $q_1 \in F$, q_0 and q_1 are distinguishable.
- similarly, $\{q_0, q_3\}, \{q_2, q_1\}, \{q_2, q_3\}$ are all distinguishable.
- q_1 and q_3 have self-loops labeled by $0, 1$, hence $\{q_1, q_3\}$ are equivalent.
- what about $\{q_0, q_2\}$?

table-filling algorithm (cont.)

table-filling algorithm

- Base case: if $p \in F$ and $q \notin F$, then the pair $\{p, q\}$ is distinguishable.
- Inductive: let $p, q \in Q, a \in \Sigma$ and $p \xrightarrow{a} r, q \xrightarrow{a} s \in \delta$; if $\{r, s\}$ is distinguishable, then $\{p, q\}$ is distinguishable.

Sketch of proof

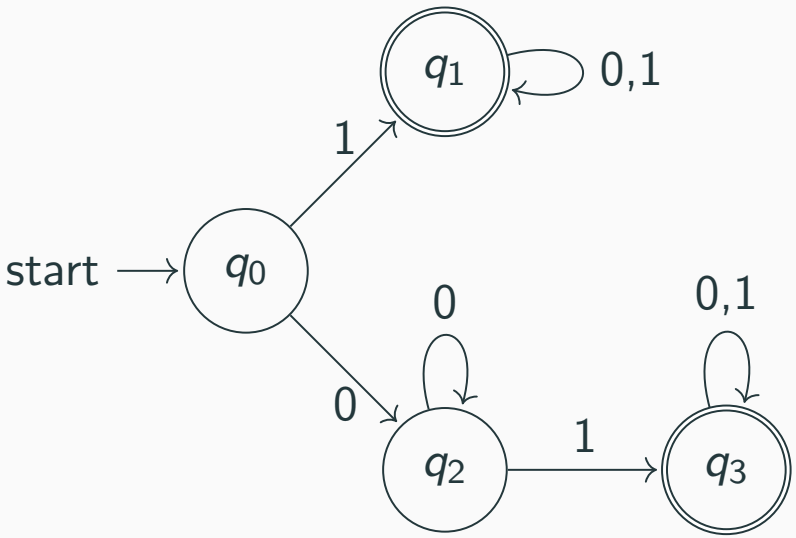
- base case is obvious.
- if $\{r, s\}$ is distinguishable, then there exists a w s.t.

$$r \xrightarrow{w} r' \in F \text{ and } s \xrightarrow{w} s' \notin F$$

(the other case is symmetric)
$\Rightarrow p \xrightarrow{aw} r' \in F$ and $q \xrightarrow{aw} s' \notin F$.
$\Rightarrow \{p, q\}$ is distinguishable. ■

table-filling algorithm (cont.)

Thm.
If two states are not distinguished by the table-filling algorithm, then they are equivalent. □



q_0				
q_1	X			
q_2		X		
q_3	X		X	
	q_0	q_1	q_2	q_3

- by algorithm, we see $\{q_0, q_2\}, \{q_1, q_3\}$ are equivalent.

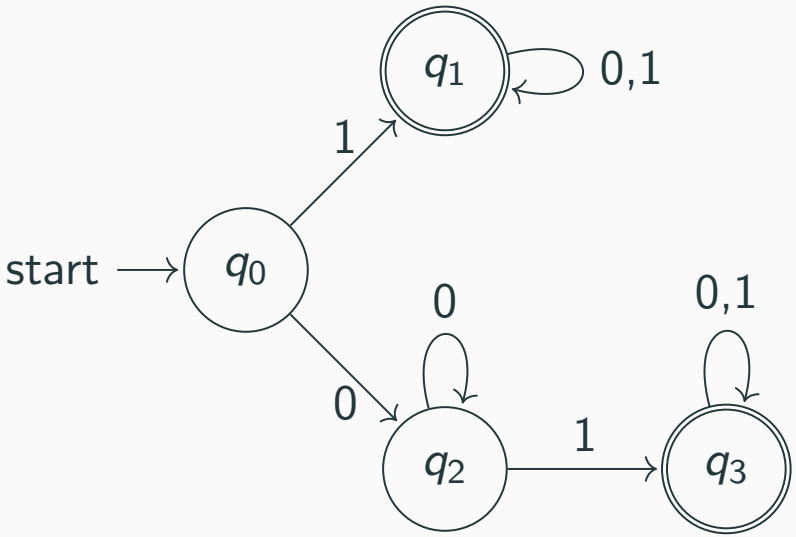
equivalence of DFAs

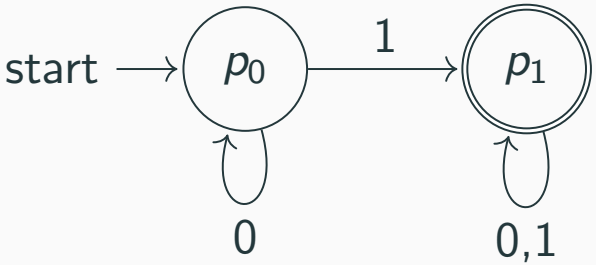
equivalence
given two regular languages L_0 and L_1, decide whether $L_0 = L_1$.

- convert L_0 and L_1 into DFAs M_0 and M_1 if necessary.
- put M_0 and M_1 together and run table-filling algorithm to see if their initial states are equivalent or not.
- if the initial states are equivalent, then $L(M_0) = L(M_1)$
  otherwise, $L(M_0) \neq L(M_1)$.

technically this union DFA has two initial states, however the initial state is irrelevant as far as testing state equivalence using table-filling algorithm.

equivalence of DFAs (cont.)





	q_0	q_1	q_2	q_3	p_0	p_1
q_0						
q_1	X					
q_2		X				
q_3	X		X			
p_0		X		X		
p_1	X		X		X	

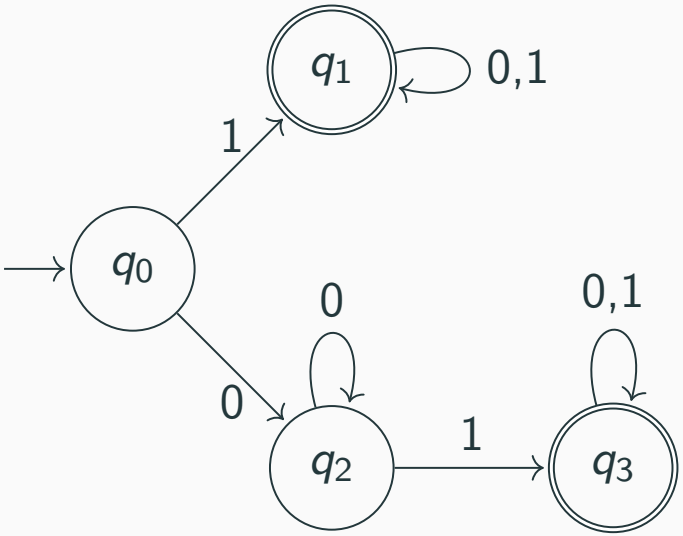
- by algorithm, we see $\{q_0, p_2\}$ is equivalent and hence both DFAs accept the same language.
- moreover, we found $\{q_0, q_2, p_0\}, \{q_1, q_3, p_1\}$ are equivalent.

minimization of DFA

minimization
given a DFA M, can one find an equivalent M' with the minimum number of states?

- surprisingly, the test for equivalence of states can solve the minimization problem.
- the algorithm is as follows:
  1. remove all the states unreachable from the initial state.
  2. run the table-filling algorithm to find equivalent states.
  3. construct M' with equivalence classes as states.

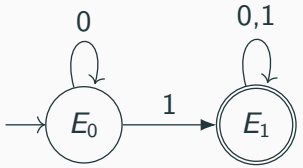
Minimization of DFA



q_0				
q_1	X			
q_2		X		
q_3	X		X	
	q_0	q_1	q_2	q_3

- by algorithm, we see $E_0 = \{q_0, q_2\}, E_1 = \{q_1, q_3\}$ are equivalent.
- $M' = (\{E_0, E_1\}, \{0, 1\}, \delta', E_0, \{E_1\})$ and

δ'	0	1
E_0	E_0	E_1
E_1	E_1	E_1

- 

minimization of DFAs

Thm.
The equivalence of states is transitive. □

Proof.
suppose not: $\{p, q\}, \{q, r\}$ are equivalent but $\{p, r\}$ is distinguishable

$\Rightarrow$ there exists w s.t. $p \xrightarrow{w} p' \in F$ but $r \xrightarrow{w} r' \notin F$

- if $q \xrightarrow{w} q' \in F$, it contradicts to $\{q, r\}$ is equivalent
- if $q' \notin F$, it contradicts to $\{p, q\}$ is equivalent

■

Cor.
The minimization algorithm is correct, namely, the DFA M' constructed from the algorithm has as few states as any of equivalent DFA. Moreover, such a minimal DFA is unique. □

minimization of DFA

There is an another point of view:

Thm. (Myhill–Nerode)
Let $L \subseteq \Sigma^*$ be a language and $w \in \Sigma^*$ be a string. Define the set

$$\mathtt{Next}_L(w) := \{t \in \Sigma^* \mid wt \in L\}$$

and let $\equiv_L$ be an equivalence relation defined by

$$u \equiv_L v \iff \mathtt{Next}_L(u) = \mathtt{Next}_L(v).$$

We have

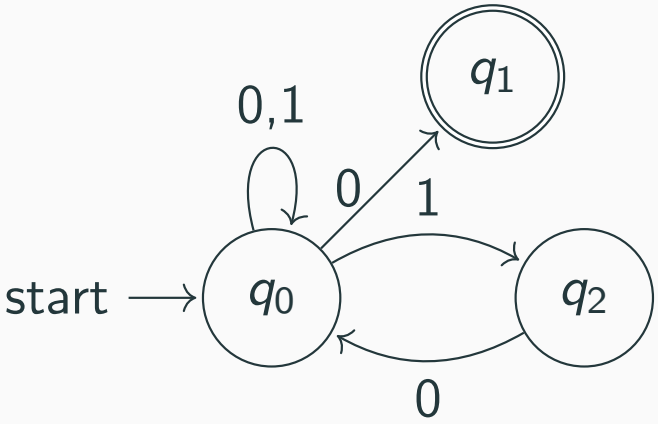
- L is regular iff the number k of equivalence classes of $\equiv_L$ is finite.
- in this case, there is a DFA M with k states such that $L(M) = L$ and there are no DFA with a strictly smaller number of states that recognizes L.

other DFA minimization algorithms

- Myhill-Nerode Theorem can be generalized to *tree automata* and other models.
- most of the minimization algorithms like Hopcroft's Algorithm or Moore's Algorithm are based on the Theorem[2].
- their efforts are on optimizing the partition refinement algorithms, e.g., Hopcroft's trick [EB23, sect. 2.2.3]
- about 50 years later, there are new approaches from functorial [CP20] and coalgebra's [JW23] viewpoints

[2] c.f. `https://en.wikipedia.org/wiki/DFA_minimization`

minimization of NFA



- one might imagine the same states-equivalence technique may work for NFAs.
- we can do that for sure; however the resulting NFA may not be the "minimum".
- consider the NFA on the left: none of these three states are equivalent.
- but we can remove q_2 and still get an NFA recognizing the same language.

Remark.

- minimization of NFA is hard (PSPACE-complete [EB23, sec. 2.3.2]).
- the minimal NFAs may not be unique!

summary

Given a regular language, we have algorithms to decide or compute:

- the union, intersection, complement, mirror of a language (exercise)
- emptiness, universality (its complement is empty)
- equivalence

What is missing?

Pumping lemma

testing whether a language is regular?

Thm.
Testing whether a context-free language is regular is undecidable[3]. □

- well, we don't have an algorithm.....
- however, we can give a less powerful criteria to prove that a language is *not regular*.

[3] c.f. [HMU07, Chap. 6]

pumping lemma

pumping lemma
Let L be a regular language over Σ. There exists $p \geq 1$ such that $\forall w \in L$ such that $|w| > p$, there exists a partition $w = xyz$, $x, y, z \in \Sigma^*$ such that

1. $|y| > 0$, i.e., $y \neq \epsilon$;
2. $|xy| < p$;
3. for each $i \geq 0$, the string $xy^i z \in L$.

What does that means?

- for every regular language L, there is a string w of length p such that, if L recognizes strings longer than p, then there must be a "loop" in the automation (or in other word, a Kleene star in the `regex`).
- therefore, there is a substring of the word (the one belong to the loop) which can be pumped (repeated arbitrary many times while staying in L).

pumping lemma (cont.)

Pumping Lemma
Let L be a regular language over Σ. There exists $p \geq 1$ such that $\forall w \in L$ such that $|w| > p$, there exists a partition $w = xyz$, $x, y, z \in \Sigma^*$ such that

1. $|y| > 0$, i.e., $y \neq \epsilon$;
2. $|xy| \leq p$;
3. for each $i \geq 0$, the string $xy^i z \in L$.

How do we use it?

- It is a *necessary condition* for a language to be regular:
  1. if a language is regular, it can be "pumped". By contrapositive, if a language cannot be "pumped", then it is not regular.
  2. if a language can be pumped, it is not necessary regular.

Example of non-regular language

Example
$L = \{a^n b^n \mid n \in \mathbb{N}_0\}$ is not regular.

Proof.

- Suppose L is regular. Let $p \geq 1$ be the pumping length given by the pumping lemma.
- Consider $w = a^p b^p \in L$. $|w| = 2p > p$. By pumping lemma, there is a partition $w = xyz$ such that $xy^i z \in L$ for $i \geq 0$.
- By lemma, $|xy| \leq p$ and $|y| > 0$, i.e., $\underbrace{a^i}_{x} \underbrace{a^j}_{y} \underbrace{a^{p-i-j} b^p}_{z}$ and $j > 0$.
- However $xy^0 z = xz = a^{p-j} b^p \notin L$ which leads to a contradiction.

■

Cor.
$C = \{w \mid w$ has an equal number of a's and b's$\}$ is not regular. □

Example of non-regular language

Example
$L = \{ww \mid w \in \{0,1\}^*\}$ is not regular.

Proof.

- Suppose L is regular. Let $p \geq 1$ be the pumping length given by the pumping lemma.
- Consider $s = 0^p 1 0^p 1$. $|s| > p$.
- By pumping lemma, there is a partition $s = xyz$ s.t. $|xy| < p$, $|y| > 0$ and $|xy^i z| \in L$ for all $i \in \mathbb{N}_0$. Thus $y \in 0^+$.
- But then $xz \notin L$.

■

example of non-regular language

Example
$L = \{0^i 1^j \mid i > j\}$ is not regular.

Proof.

- Suppose L is regular and p is the pumping length. Choose $s = 0^{p+1}1^p$.
- By pumping lemma, there is a partition $s = xyz$ s.t. $|xy| < p$, $|y| > 0$ and $|xy^i z| \in L$ for all $i \in \mathbb{N}_0$.
- Since $|xy| \leq p$, $y \in 0^+$. But $xz \notin L$ for $|y| > 0$. Contradiction.

■

pumping lemma (cont.)

pumping lemma

Let L be a regular language over Σ. There exists $p \geq 1$ such that $\forall w \in L$ such that $|w| > p$, there exists a partition $w = xyz$, $x, y, z \in \Sigma^*$ such that

1. $|y| > 0$, i.e., $y \neq \epsilon$;
2. $|xy| \leq p$;
3. for each $i \geq 0$, the string $xy^i z \in L$.

Proof.

- Let $M = (Q, \Sigma, \delta, q_0, F)$ be a `DFA` s.t. $L(M) = L$. Set $p = |Q|$.
- For any string $w = a_1 \cdots a_n \in L(M)$ with $n \geq p$, let $r_1, s_1, \ldots, s_n, r_{n+1}$ be the corresponding run (i.e. $r_1 = q_0$, $r_{i+1} = \delta(r_i, s_i)$ for all $i \in [n]$).
- Since $n + 1 > n \geq p = |Q|$, there are $1 \leq j < k \leq p + 1$ s.t. $r_j = r_k$.
- Choose $x = a_i \cdots a_{j-1}, y = a_j \cdots a_{k-1}$ and $z = a_k \cdots a_n$.
- Thus, $xy^i z \in L(M)$ for $i \geq 0$. Since $j \neq k$, $|y| > 0$. $k \leq p + 1$ so $|xy| \leq p$.

Summary

Given a regular language, we have algorithms to decide or compute:

- the union, intersection, complement, mirror of a language (exercise)
- emptiness, universality (its complement is empty)
- equivalence

Given a language,

- it is impossible to decide whether or not it is regular.
- however, pumping lemma gives us a criteria to prove that a language is not regular.

What should we remember?

- everything! all these properties translate to trees and tree automata.
- for all other formalism (Grammar, Pushdown automata,..etc), some of these properties hold, others don't.
- when they do hold, often the proofs are similar.

Where to go?

- More formal models, e.g. Context-free grammar, Turing machine,...etc.
- To infinite and beyond, e.g., ω-Automata or Büchi Automata.
- It is widely studied in computational complexity, formal verification and natural language processing.

to infinity and beyond

ω-automata

we would like to generalize inputs to finite automata

instead of finite input strings, let us consider an infinite input strings over Σ

$$\alpha = a_1 a_2 \cdots a_n \cdots$$

let $M = (Q, \Sigma, \delta, q_0, F)$ be a finite automation, we would like to define a run $\rho = q_0 a_1 q_1 a_2 q_2 \ldots$ on α to be an infinite sequence such that

$$\forall i \geq 0.\ q_i \xrightarrow{a_i} q_{i+1} \in \delta$$

Problem: there is no "final" state in an infinite run

What is an accepting run?

Büchi acceptance

let us define

$$\mathrm{Inf}(\rho) \stackrel{\text{def}}{=} \{q \in Q \mid q \text{ occurs infinitely many times in } \rho\}$$

Defn. (Büchi acceptance)

- An infinite run ρ of $M = (Q, \Sigma, \delta, q_0, F)$ on α is accepting if $\mathrm{Inf}(\rho) \cap F \neq \emptyset$
- An infinite string α is accepted by M if there exists an accepting run on α
- the ω-language of M, denoted by $L_\omega(M)$ is defined as

$$L_\omega(M) \stackrel{\text{def}}{=} \{\alpha \mid \alpha \text{ is an infinite string accepted by } M\}$$

example

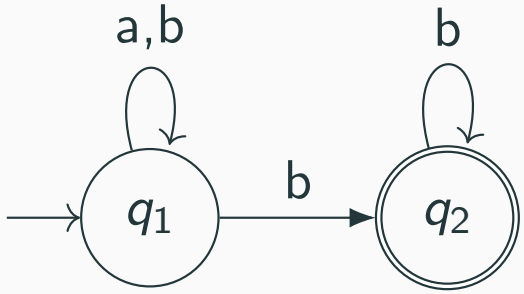


Figure 19: NFA N_6

we have

$$L_\omega(N_6) = \{\alpha \mid \alpha \text{ has only finitely many } a\text{'s}\}$$

the corresponding ω-regular expression is $(a|b)^* b^\omega$

nondeterminism

for finite automata over finite input strings, we know nondeterminism does not give us more expressive power

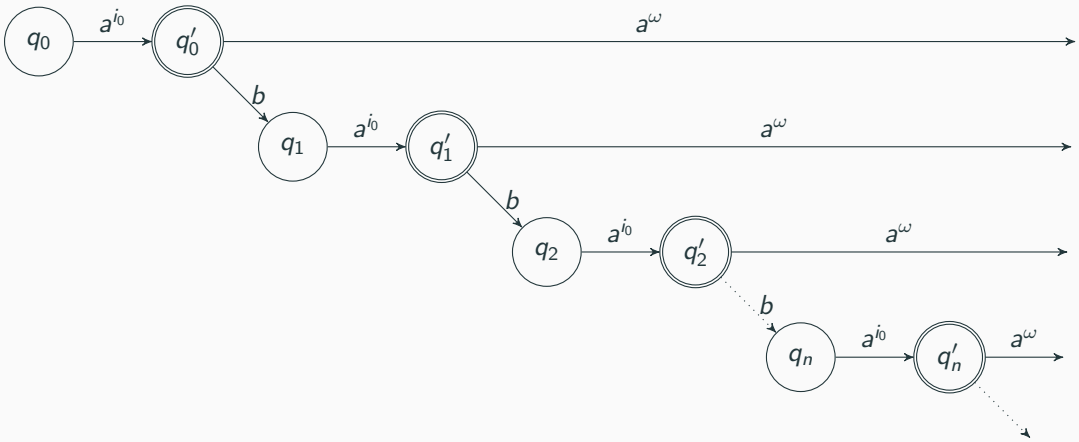
however,

Thm.
Nondeterministic Büchi automata recognize strictly more language than deterministic ones □

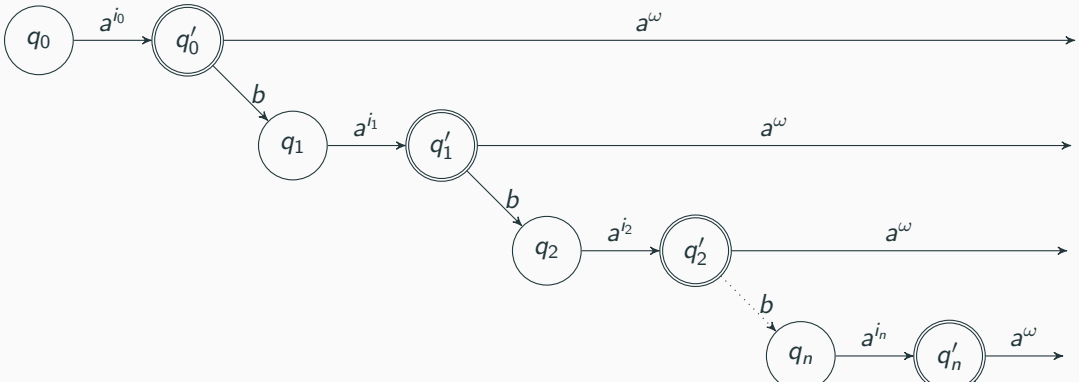
Example
the ω-language defined by Fig. 19 cannot be accepted by any deterministic Büchi automata

sketch of the proof

Let $\mathcal{A} = (Q, \{a, b\}, \delta, q_0, F)$ be a DBA with $|Q| = n$ such that $L_\omega(\mathcal{A}) = L_\omega(N_6)$.



- consider $w_0 = a^\omega \in L_\omega(N_6)$.
- by assumption there is a run ρ_0 on w_0 which visits a state in F infinitely many times.
- let q'_0 be the first state in F reached by the run ρ_0 after reading i_0 letters.



- consider the word $w_1 = a^{i_0} b a^\omega$.
- since A is deterministic, the run ρ_1 on w_1 and the one ρ_0 on w_0 are equal up to q'_0.
- $w_1 \in L_\omega(N_6)$, there must be a transition from q'_0 to a state q_1 different from q_0, q'_0.

the class of ω-regular language

Defn.

- A language L_ω is called ω-regular if there is a nondeterministic Büchi automata N such that $L_\omega(N) = L_\omega$
- denote by $\mathcal{R}_\omega$ the class of ω-regular languages

□

Remark.

- $\mathcal{R}_\omega$ is closed under boolean operations
- emptiness of Büchi automata is decidable in linear time (cf. [BK08, sec. 5.2])
- $\mathrm{LTL} \subsetneq \mathcal{R}_\omega$ (c.f. [VW84] or [BK08, Thm 5.41, Remark 5.43])

□

conclusion remark

where to go

- automata theory is a rich field
- it is widely studied in computational complexity, formal verification, and natural language processing
- you will see applications of automata theory in formal verification next week

References i


Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman.
The design and analysis of computer algorithms.
Addison-Wesley, Reading, 1974.


Christel Baier and Joost-Pieter Katoen.
Principles of model checking.
The MIT Press, Cambridge, Mass, 2008.
OCLC: ocn171152628.


Janusz A. Brzozowski.
Derivatives of regular expressions.
J. ACM, 11(4):481–494, October 1964.


Thomas Colcombet and Daniela Petrişan.
Automata Minimization: a Functorial Approach.
Logical Methods in Computer Science, Volume 16, Issue 1, March 2020.

References ii

J. Esparza and M. Blondin.

Automata theory: An algorithmic approach.

The MIT Press, 2023.

John E. Hopcroft, Rajeev Motwani, and Jeffrey D. Ullman.

Introduction to automata theory, languages, and computation.

Pearson/Addison Wesley, Boston, 3rd ed edition, 2007.

OCLC: ocm69013079.

Jules Jacobs and Thorsten Wißmann.

Fast coalgebraic bisimilarity minimization.

Proc. ACM Program. Lang., 7(POPL), 2023.

References iii

Michael Sipser.

Introduction to the theory of computation.

Cengage Learning, Australia Brazil Japan Korea Mexiko Singapore Spain United Kingdom United States, third edition, international edition edition, 2013.

Ian Erik Varatalu, Margus Veanes, and Juhan Ernits.

Re#: High performance derivative-based regex matching with intersection, complement, and restricted lookarounds.

Proc. ACM Program. Lang., 9(POPL), January 2025.

References iv

Moshe Y. Vardi and Pierre Wolper.
Automata theoretic techniques for modal logics of programs: (extended abstract).
In *Proceedings of the Sixteenth Annual ACM Symposium on Theory of Computing*, STOC '84, page 446–456, New York, NY, USA, 1984. Association for Computing Machinery.

Questions?