COMPSCI 350: Automata

Cristian S. Calude

Semester 1, 2018

R. Hamming

"The purpose of computing is insight, not numbers."

Bibliography

- M. Sipser. *Introduction to the Theory of Computation*, PWS 1997. (textbook)

- T. H. Cormen, C. E. Leiserson, R. L. Rivest, C. Stein, *Introduction to Algorithms* MIT Press and McGraw-Hill, 2011. 2nd ed.
- Regex simulator, https://regex101.com.

Assignment 1 : Friday 23 March 2018 before 11.55pm, submitted via Canvas; worth **5%.**

Midterm Test : 26 March 2018, in class, time: 15:00-16:00 in OGHLLecTh/102-G36 and Conf. Centre Lecture Theater/423-342; worth **30%.**

Mathematical background

Finite machines

DFA

NFA

Minimisation of DFAs

Beyond regular languages

Pattern matching

A *set* is a group of elements represented as a unit. The objects in a set are called *elements* or *members*. The order of elements is irrelevant; repetition of its members does not matter.

Sets can be finite, like $\{1, 3, 5\}$, or infinite, like $\{1, 3, 5, 7, 11, 13, \dots\}$.

The symbols $\in$ and $\notin$ denote set membership and non-membership, respectively. For example $7 \notin \{1, 3, 5\}$ and $7 \in \{2, 3, 5, 7, 11, 13, \dots\}$.

Two sets A, B are equal, written $A = B$, if they have the same elements; otherwise, $A \neq B$. We say that A is a *subset* of B, written $A \subseteq B$ if every element of A is also in B; A is a *proper subset* of B, written $A \subset B$ if $A \subseteq B$ and $A \neq B$

The set of natural numbers is $\mathbf{N} = \{0, 1, 2, 3, \dots\}$. The set with no elements is called the *empty set* and is denoted by $\emptyset$.

Sets can be described by a specific property P, $\{n \mid P(n)\}$. For example, $\{n \in \mathbf{N} \mid n \text{ is prime}\} = \{2, 3, 5, 7, 11, \dots\}$.

Sets can be combine with various operations, including *union* ($A \cup B$ consists of all elements in A or in B), *intersection* ($A \cap B$ consists of all elements in A and in B), *complement* ($\overline{A}$ consists of all elements not in A) and *power set* (2^A consists of all subsets of elements of A).

$$\{3, 4\} \cup \emptyset = \{3, 4\},$$

$$\{5, 1, 10\} \cap \{2, 3, 5, 7, 11, \dots\} = \{5\},$$

$$\overline{\{2, 3, 5, 7, 11, \dots\}} = \{1, 4, 6, 8, \dots\},$$

$$2^{\{0,1\}} = \{\emptyset, \{0\}, \{1\}, \{0, 1\}\}.$$

$$\text{For every set } A, A \cap \overline{A} = \emptyset.$$

$$\text{For every set } A, 2^A \neq \emptyset.$$

A *function* (or a *mapping*) is a rule/process that takes an input and produces an output. *For every function, the same input always produces the same output.*

If f is a function that produces the output b on input a we write $f(a) = b$.

The set of inputs for a function f is called the *domain* (D) of f; the sets of outputs is called the *range* (R) of f. We write $f : D \to R$.

The function $f : D \to R$ is

- *injective* if for every $x \neq y$ in D, $f(x) \neq f(y)$;
- *surjective, or onto* if for every $z \in R$ there exists $x \in D$ such that $f(x) = z$;
- *bijective* if it is both injective and surjective.

The function $f : \{0, 1, 2, 3, 4\} \to \{0, 1, 2, 3, 4\}$ defined by

- $f(n) = 0$ for all $n \in \{0, 1, 2, 3, 4\}$ is not injective and not surjective;
- $f(n) = n + 1$ for $n \in \{0, 1, 2, 3\}$ and $f(4) = 0$ is bijective;

No function $f : \{0, 1, 2, 3, 4\} \to \{0, 1, 2, 3, 4\}$ can be injective but not surjective (or surjective but not injective).

The function $f : \{0, 1, 2, 3, 4\} \to \{0, 1, 2, 3, 4, \dots\}$ defined by $f(n) = n$ for all $n \in \{0, 1, 2, 3, 4\}$ is injective and not surjective.

The function $f : \{0, 1, 2, 3, \dots\} \to \{0, 1, 2\}$ where $f(n)$ is the reminder of the division of n by 3 for all $n \in \{0, 1, 2, 3, \dots\}$ is surjective and not injective.

A *sequence* is a list of elements in some order. We use parentheses to describe sequences like in $(4, 1, 44)$. As the order is important, $(4, 1, 44) \neq (44, 1, 4)$.

Finite sequences are also called *tuples*. A tuple with k elements is called k-tuple; if $k = 2$, we call it a *pair*.

The *cross product* of the sets A, B is defined by

$$A \times B = \{(a, b) \mid a \in A, b \in B\}.$$

For example,

$$\{a, b, c\} \times \{0, 1\} = \{(a, 0), (b, 0), (c, 0), (a, 1), (b, 1), (c, 1)\}.$$

A subset R of a set $A \times B$ is called a *(binary) relation*.

An *equivalence* relation $R \subseteq A \times A$ (also denoted by $\equiv$) has the following three properties:

1. *reflexivity*: for every $x \in A$, $(x, x) \in R$,
2. *symmetry*: for every $x, y \in A$, if $(x, y) \in R$, then $(y, x) \in R$,
3. *transitivity*: for every $x, y, z \in A$, if $(x, y) \in R$ and $(y, z) \in R$, then $(x, z) \in R$.

We now **prove** that the relation $n \equiv m$ defined on natural numbers by "$n - m$ is a multiple of 7" is an equivalence relation.

First, we have $n \equiv n$ because 7 divides $n - n = 0$. Second, if $n \equiv m$ then (by definition) $n - m$ is a multiple of 7, so $m - n = -(n - m)$ is also a multiple of 7. Third, if $n \equiv m$ and $m \equiv t$, then (by definition) $n - m$ and $m - t$ are multiples of 7, so $m - t = (m - n) + (n - t)$ is also a multiple of 7 because the sum of two multiples of 7 is also a multiple of 7.

A *predicate* or *property* is a function $P : A \to \{\mathrm{TRUE}, \mathrm{FALSE}\}$. Sometimes we write $P : A \to \{0, 1\}$, where 0 stands for FALSE and 1 stands for TRUE.

For example, the predicate PRIME: $\{1, 2, 3, \dots\} \to \{0, 1\}$ is defined by PRIME(n)=0, if n is composite and PRIME(n)=1, if n is prime.

PRIME(13)=1, PRIME($2^{77,232,917} - 1) = 1$ (actually, this is the largest known prime as of January 2018), PRIME($2^{77,232,917}) = 0$.

An *alphabet* is a finite set. The elements of an alphabet are called *symbols*. Alphabets are usually denoted by capital (sometimes Greek) letters:

$$\Sigma = \{a\}, B = \{0, 1\}, \Gamma = \text{the set of 7-bit ASCII characters.}$$

A *string* over an alphabet is a finite sequence of symbols over the alphabet. For example, 1000 is a string over the alphabet B. The *length* of the string w over the alphabet Σ – denoted by $|w|$ – is the number of symbols it contains. The length of 00001 is 5. The string of length zero is called the *empty string* and is denoted by ε. Strings can be concatenated: from x and y get xy; $|xy| = |x| + |y|$.

The set of all strings over the alphabet Σ is denoted by Σ^*. A string x is a *substring* of y if there exist two strings u, v such that $y = uxv$: *cad* is a substring of *abracadabra* over the alphabet $\{a, b, c, d, r\}$.

The *lexicographical order* of strings is defined in two steps: a) a shorter string precedes a longer string, and b) strings of the same length are ordered as in the dictionary (this assumes an ordering of the symbols in the alphabet).
If $B = \{0, 1\}$ and 0 precedes 1, then we have

$$\varepsilon < 0 < 1 < 00 < 01 < 10 < 11 < 000 < 001 < \dots$$

A *language* is a set of strings. All set-theoretic operations can be applied to languages, but there are specific language-theoretic operations like *concatenation*:

$$AB = \{xy \mid x \in A, y \in B\}.$$

The values TRUE and FALSE are called *Boolean values* and are denoted by 1 and 0, respectively. The following operations with Boolean values are important:

- Negation (NOT): $\neg x = 1 - x$,
- Disjunction ($\vee$): $x \vee y = \max\{x, y\}$,
- Conjunction ($\wedge$): $x \wedge y = \min\{x, y\}$,
- Implication ($\rightarrow$): $x \rightarrow y = \neg(x) \vee y = \max\{1 - x, y\}$,
- Equivalence ($\leftrightarrow$): $x \leftrightarrow y = (x \rightarrow y) \wedge (y \rightarrow x)$,
- Exclusive OR ($\oplus$): $\oplus(x, y) = \neg(x \leftrightarrow y)$.

The two most common quantifiers are "for all" – $\forall$ and "there exists" – $\exists$. If P is a predicate, then

- $\forall x P(x)$ means "for all x, $P(x)$ is true.
- $\exists x P(x)$ means "there exists x such that $P(x)$ is true.

Informal	Formal
For each natural number n, $n \cdot 2 = n + n$.	$\forall n \in \mathbf{N}\ (n \cdot 2 = n + n)$.
For some natural number n, n^2 is equal to 25.	$\exists n \in \mathbf{N}\ (n^2 = 25)$.

The following important rules relate negation to quantifiers:

$$\neg(\forall x P(x)) = \exists x(\neg P(x)),$$

$$\neg(\exists x P(x)) = \forall x(\neg P(x)).$$

Informal.
All horses fly. Negation (All horses fly) = There is a horse that does not fly.
Formal.

$$\forall x(\text{horse}(x) \to \text{fly}(x)).$$

$$\begin{aligned}\neg(\forall x(\text{horse}(x) \to \text{fly}(x))) &= \exists x(\neg(\text{horse}(x) \to \text{fly}(x))) \\ &= \exists x(\text{horse}(x) \wedge \neg\text{fly}(x))\end{aligned}$$

because $\neg(A \to B) = A \wedge \neg B$.

"Theorems and proofs are the heart and soul of mathematics and definitions are its spirit" says Sipser.

Definitions describe clearly and precisely the objects and notions we use.

Mathematical statements express properties of defined objects. They may be true or false, but they always have to be *precise*.

A *proof* is a convincing – ideally, in an absolute sense – argument that a statement is true. It should not only be "beyond reasonable doubt", but "beyond any doubt".

A *theorem* is a mathematical statement proved to be true. A *lemma* is a proved mathematical statement useful in the proof of a more important mathematical statement. A *corollary* is a proved mathematical statement which can easily derived from another mathematical statement, usually a theorem.

The only way to show the truth or falsity of a mathematical statement is via a *mathematical proof*. Finding proofs is not easy, even if we use a *proof-assistant* (like Isabelle or Coq), i.e. a sophisticated software designed to assist with the development of formal proofs by human-machine collaboration.

A proof is typically a formal argument showing the truth of an implication of the form "P implies Q". A proof of an equivalence is a proof of both implications "P implies Q" and "Q implies P".

In what follows we shall present some typical examples of proofs: they will appear in a form or another in what follows.

Theorem 0.10 *For any two sets A and B,*

$$\overline{A \cup B} = \overline{A} \cap \overline{B}.$$

Proof. The theorem states that two sets are equal, hence we need to prove that every element in $\overline{A \cup B}$ is in $\overline{A} \cap \overline{B}$ and, conversely, every element in $\overline{A} \cap \overline{B}$ is in $\overline{A \cup B}$.

If $x \in \overline{A \cup B}$, then $x \notin A \cup B$ (by the definition of the complement), hence $x \notin A$ and $x \notin B$ (by the definition of the union), so $x \in \overline{A}$ and $x \in \overline{B}$ (by the definition of the complement), which means that $x \in \overline{A} \cap \overline{B}$ (by de definition of the intersection). This shows that $\overline{A \cup B} \subseteq \overline{A} \cap \overline{B}$.

Next we shall prove the converse implication, i.e. $\overline{A} \cap \overline{B} \subseteq \overline{A \cup B}$. Try it!

A number is *rational* if it is a ratio of two integers, $\frac{n}{m}$, where $m \neq 0$.

Theorem. *There exist irrational numbers a and b such that a^b is rational.*

Non-constructive proof. The number $\sqrt{2}^{\sqrt{2}}$ is either rational **or** irrational. If it is rational, our statement is proved: $a = b = \sqrt{2}$. If it is irrational, then take $a = \sqrt{2}^{\sqrt{2}}$, $b = \sqrt{2}$ and compute: $a^b = (\sqrt{2}^{\sqrt{2}})^{\sqrt{2}} = 2$. The statement was proved. This proof is non-constructive because we don't know whether $\sqrt{2}^{\sqrt{2}}$ is rational or not.

Theorem. *There exist irrational numbers a and b such that a^b is rational.*

Constructive proof. The numbers $a = \sqrt{2}$, $b = \log_2 9$ are irrationals and $a^b = 3$ is rational. The statement was proved.

Really? A simple analysis of the proof shows that in fact we have an implication:

If the numbers $a = \sqrt{2}$, $b = \log_2 9$ are irrationals,
then $a^b = 3$ is rational.

To prove the theorem we need to prove that the implication is true. As the conclusion is true, we need to show that the hypothesis is true, that is two facts: a) $\sqrt{2}$ is irrational and b) $\log_2 9$ is irrational!

Theorem 0.14 $\sqrt{2}$ *is irrational.*

Proof. Assume by absurdity that $\sqrt{2}$ is rational, that is,

$$\sqrt{2} = \frac{N}{M},$$

where $M \neq 0$. If both N, M are divisible by the same integer t, then divide them by t; the value of the fraction will not change. Continue this (finite!) process till no such integer exists, so

$$\sqrt{2} = \frac{N}{M} = \frac{n}{m}.$$

Both n, m cannot be even. As $m \neq 0$, we can write $m\sqrt{2} = n$ and by squaring both members we get

$$2m^2 = n^2. \tag{1}$$

Continuation of the proof. From (1) we deduce that n^2 is even, so n is also even as the square of an odd number is odd. So, there exists an integer k such that $n = 2k$. Substituting in the equation (1) we get:

$$2m^2 = (2k)^2 = 4k^2.$$

This mean that $m^2 = 2k^2$, that is, m is even, a contradiction!

Theorem. $\log_2 9$ *is irrational.*

Proof. Assume by absurdity that $\log_2 9$ is rational, that is $\log_2 9 = \frac{n}{m}$, where n, m are integers and $m \neq 0$. By the properties of logarithms, 9^m would be equal to 2^n, a contradiction because the former is odd, and the latter is even.

Proof by induction is a method to show that all elements of an infinite countable set have a certain property.

Consider a property $P(i)$ of natural numbers; the goal it to show that $P(i)$ is true for every natural number i. As there are infinitely many i's, we cannot verify individually each of them, so the proof by induction comes handy.

The proof by induction consists in two steps:

1. *Basis*: Prove that $P(k)$ is true for a fixed natural number k.
2. *Induction step*: For each $i \geq k$ assume that $P(i)$ is true – the *induction hypothesis* –, and prove that $P(i + 1)$ is also true.

Theorem. $1 + 2 + 3 + \cdots + n = \frac{n(n+1)}{2}$.

Proof. For the basis we take $k = 1$: $1 = \frac{1 \cdot 2}{2}$ checks. Then, we assume that for every $i \geq k = 1$ we have

$$1 + 2 + 3 + \cdots + i = \frac{i(i+1)}{2}, \qquad (2)$$

and we need to prove that

$$1 + 2 + 3 + \cdots + (i+1) = \frac{(i+1)(i+2)}{2}.$$

Indeed, using the induction hypothesis (2) we get:

$$1 + 2 + 3 + \cdots + i + (i+1) = (1 + 2 + 3 + \cdots + i) + (i+1)$$

$$= \frac{i(i+1)}{2} + (i+1) = \frac{(i+1)(i+2)}{2}.$$

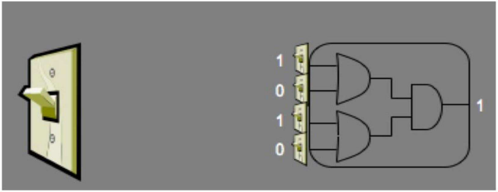
Question

Are there finite memory machines accepting as input finite binary sequences of any length and deciding whether the sequence has a certain property (for example, it has an even number of 0's)?

Using "states" to remember the 'property' seems a good idea, but don't we have to keep adding newer and newer 'states' as the input gets longer and longer?

Re-phrased: Is a finite memory enough? In general the answer seems to be negative, but ...

Probably the simplest finite machine operates a switch as follows:



So, if the switch is down, then the light goes on and if the switch is up, then the light goes off.

To this device, the switch position is an input and the light on/off is the output. The machine works with finitely many "states" for any sequence of modifications of the switch.

A *deterministic finite automaton* (*DFA*, for short) is a five-tuple $M = (Q, \Sigma, \delta, s, F)$ where

1. Q is the finite set of machine states
2. Σ is the finite input alphabet
3. δ is a transition function from $Q \times \Sigma$ to Q
4. $s \in Q$ is the start state
5. $F \subseteq Q$ is the accepting (final/membership) states.

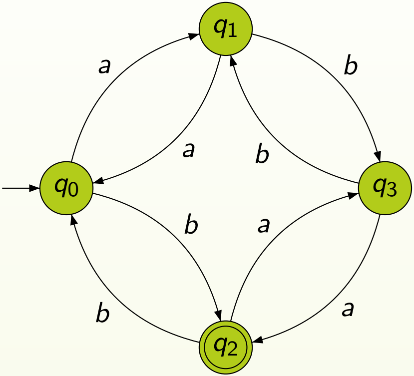
DFA: example 1

$M = (Q, \Sigma, \delta, s, F)$:

$Q = \{q_0, q_1, q_2, q_3\}$
$\Sigma = \{a, b\}$

δ	Σ	
Q	a	b
q_0	q_1	q_2
q_1	q_0	q_3
q_2	q_3	q_0
q_3	q_2	q_1

$s = q_0$
$F = \{q_2\}$



Let $M = (Q, \Sigma, \delta, s, F)$ be a DFA and $w = w_1 w_2 \cdots w_n$ be a string over Σ.

- The *trace* (*path*) of the computation of w on M is the (*unique*) sequence of states

$$s_1, s_2, \cdots, s_n, s_{n+1}$$

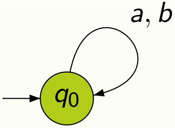
such that

$$s_1 = s, \delta(s_1, w_1) = s_2, \ldots, \delta(s_{n-1}, w_{n-1}) = s_n, \delta(s_n, w_n) = s_{n+1}.$$

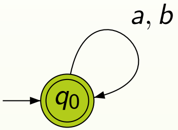
- The string w is *accepted* (or *recognised*) by M if $s_{n+1} \in F$; otherwise, w is rejected by M.
- The *language* accepted by M, denoted by $L(M)$, is the set of all accepted strings by M; if $A = L(M)$, for some DFA M, then A is called *regular*.

- Given a DFA M, check which strings M accepts.
- Given a language (set of strings) can we build a DFA M that recognises just them? If the answer is affirmative can we construct a minimal (in the sense of the number of states) DFA recognising the language?
- Which properties of DFAs can be checked algorithmically?

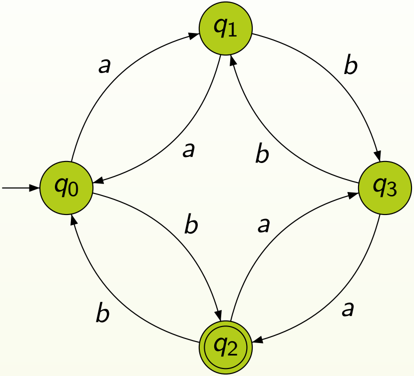
The language accepted by this DFA is empty, i.e. the DFA accepts no string.



The language accepted by this DFA consists of all strings over $\Sigma = \{a, b\}$, i.e. the language $\Sigma^* = \{a, b\}^*$.

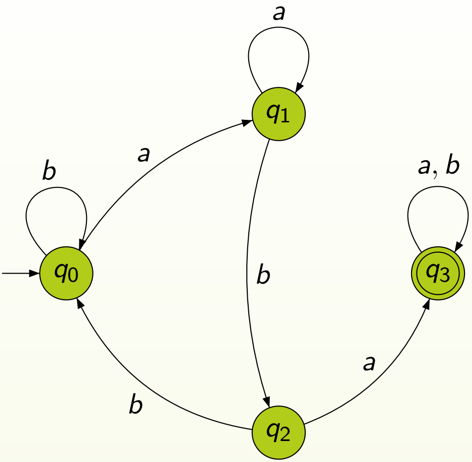


The language accepted by this DFA consists of all strings over $\Sigma = \{a, b\}$ which contain an even number of a's and an odd number of b's.

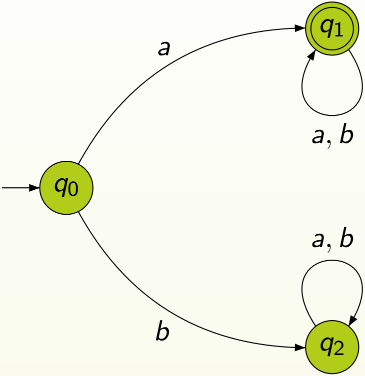


DFA: example 4

The language accepted by this DFA consists of all strings over $\Sigma = \{a, b\}$ which contain the substring aba, i.e. all the strings of the form $uabav$ with $u, v \in \{a, b\}^*$.

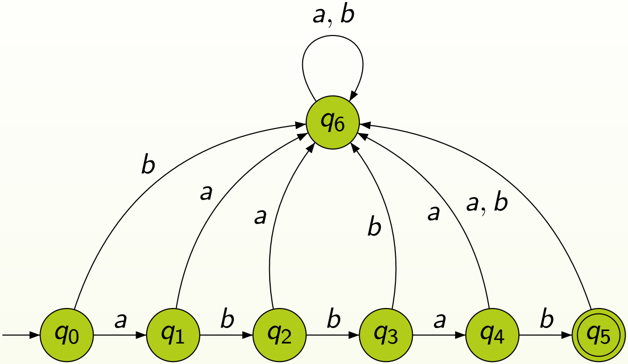


The language accepted by this DFA consists of all strings over $\Sigma = \{a, b\}$ which start with a, i.e. all the strings of the form av, with $v \in \Sigma^* = \{a, b\}^*$.

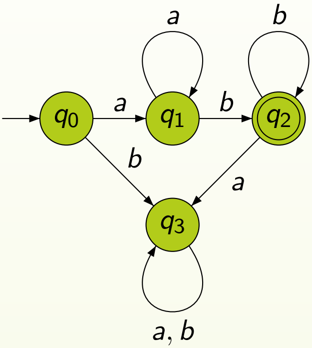


DFA: example 6

The language accepted by this DFA consists of only one string over $\Sigma = \{a, b\}$, namely *abbab*.



The language accepted by this DFA is $\{a^m b^n \mid m, n > 0\}$, where a^m means $aa \cdots a$ (m times).



The language

$$L = \{a^n b^n \mid n > 0\}$$

is not accepted by any DFA.

Why?

Informally, because a DFA can 'count' only up to the number of its states.

More formally, because, if n is greater than the number of states of a DFA supposed to accept L, then any trace (path) labelled by a^n passes twice through some state. That is, there are strings a^i and a^j for $i < j \leq n$ that fall into the same state. Thus both $a^i b^i$ and $a^j b^i$ are accepted/rejected which contradicts the definition of L.

- The complement of a regular language is also regular.
  Proof: if $A = L(M)$, where $M = (Q, \Sigma, \delta, s, F)$, then its complement, $\overline{A} = L(M')$, where $M' = (Q, \Sigma, \delta, s, \overline{F})$.
- It is algorithmically decidable whether a DFA M accepts the empty string.
  Proof: If $M = (Q, \Sigma, \delta, s, F)$, then $\varepsilon \in L(M)$ if and only if $s \in F$.
- It is algorithmically decidable whether a DFA M accepts a string w.
  Proof: Construct the trace of the computation of w on M and check whether its last state is final.

- It is algorithmically decidable whether a DFA M accepts no string.

  Proof: Given the DFA M check whether there is a path from the initial state s (has a trace of a computation) to a final state in F. We have: $L(M) = \emptyset$ if and only if there is no path from the initial state to a final state.

- It is algorithmically decidable whether a DFA M accepts infinitely strings.

  Proof: Given the DFA M, $L(M)$ is infinite if and only if there is a path from the initial state (has a trace of a computation) s to a final state in F having the following additional property: some state q in the path possesses a loop, i.e. there is a path from q to q.

The *reverse* of a string

$$w = c_1 c_2 c_3 \cdots c_n$$

is the string

$$R(w) = c_n c_{n-1} \cdots c_2 c_1.$$

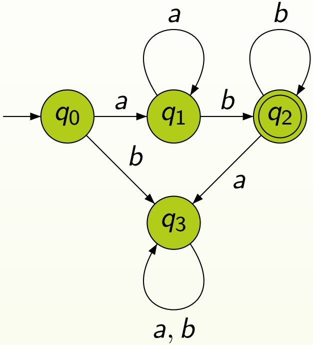
For example, $R(abaaa) = aaaba$, $R(abba) = abba$, $R(bac) = cab$.

The *reverse* of a language A is the language

$$R(A) = \{R(w) \mid w \in A\}.$$

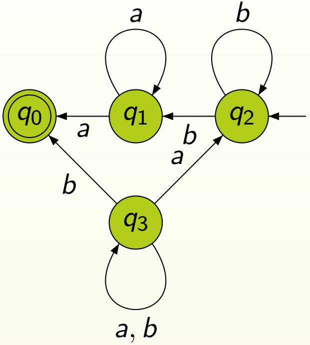
Problem: Is $R(A)$ regular whenever A is regular?

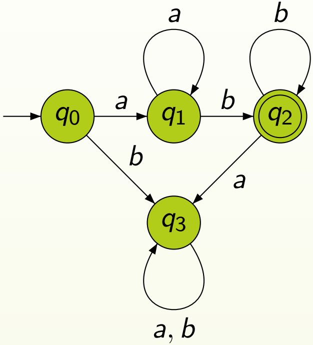
The language accepted by the DFA M is
$A = \{a^m b^n \mid m, n > 0\}$.
Is
$R(A) = \{b^n a^m \mid m, n > 0\}$
regular?

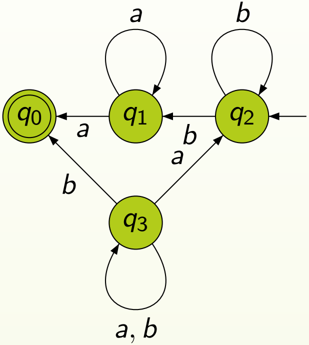


A possible solution?

Is $R(A) = \{b^m a^n \mid m, n > 0\}$ accepted by this machine, M'?




a
b
q_0
a
q_1
b
q_2
b
a
q_3
a, b


a
b
q_0
q_1
q_2
a
b
a
b
q_3
a, b

What did we do, in more general terms?

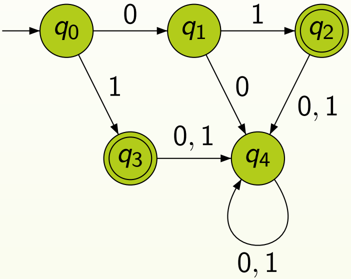
1. The initial state of M becomes the accept state of M'.
2. Every accept state of M becomes an initial state of M'.
3. If $\delta(q_1, c) = q_2$ is in M then $\delta(q_2, c) = q_1$ is in M'. That is, all transitions are reversed.

Do we have a problem with M'?
Answer: yes: M' is not a DFA!

Still, the procedure seems reasonable!

What should we do? Well, let's examine another example.

Transforming this DFA M into M' produces:
a) two initial states: q_2, q_3
b) multiple transitions with the same label (e.g. $\delta(q_4, 0) = \{q_1, q_2, q_3, q_4\}$)



Should we abandon the transformation $M \to M'$?

No. We turn it into a new concept!

A *nondeterministic finite automaton* (*NFA*, for short) is a five-tuple $N = (Q, \Sigma, \delta, S, F)$ where

1. Q is the finite set of machine states
2. Σ is the finite input alphabet
3. δ is a function from $Q \times \Sigma$ to 2^Q, the set of subsets of Q
4. $S \subseteq Q$ is a set of start (initial) states
5. $F \subseteq Q$ is the accepting (final/membership) states.

Informally, an NFA accepts a string w if there exists a (nondeterministic) trace (path) following the transition function δ on input w from an initial state to an accept state.

Let $N = (Q, \Sigma, \delta, S, F)$ be a NFA and $w = w_1 w_2 \cdots w_n$ be a string over Σ.

- A *trace* (*path*) of a computation of w on N is a sequence of states

  $$s_1, s_2, \cdots, s_n, s_{n+1}$$

  such that

  $$s_2 \in \delta(s_1, w_1), \ldots, s_n \in \delta(s_{n-1}, w_{n-1}), s_{n+1} \in \delta(s_n, w_n).$$

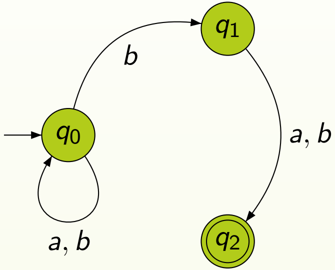
- The string w is *accepted* (or *recognised*) by N if there is a trace $s_1, s_2, \cdots, s_n, s_{n+1}$ labelled by w such that $s_1 \in S$ and $s_{n+1} \in F$; otherwise, w is rejected by N.
- The *language* accepted by N, denoted by $L(N)$, is the set of all accepted strings by N.

- The state transition function δ is more general for NFAs than DFAs. Besides having transitions to multiple states for a given input symbol, we can have $\delta(q, c)$ empty (undefined) for some $q \in Q$ and $c \in \Sigma$. This means that that we can design automata such that no state moves are possible for when in some state q and the next character read is c (that is, the human designer does not have to worry about all cases).
- Every DFA can be viewed as a special case of an NFA.

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

δ States	Σ a	b
q_0	$\{q_0\}$	$\{q_0, q_1\}$
q_1	$\{q_2\}$	$\{q_2\}$
q_2	$\emptyset$	$\emptyset$

$S = \{q_0\}$
$F = \{q_2\}$



- The string *aba* is accepted: there are two traces,

$$q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_0,$$

$$q_0 \xrightarrow{a} q_0 \xrightarrow{b} q_1 \xrightarrow{a} q_2$$

- The string *baa* is not accepted: there are two traces,

$$q_0 \xrightarrow{b} q_0 \xrightarrow{a} q_0 \xrightarrow{a} q_0,$$

$$q_0 \xrightarrow{b} q_1 \xrightarrow{a} q_2 \xrightarrow{a} ?$$

- The language accepted by this NFA is

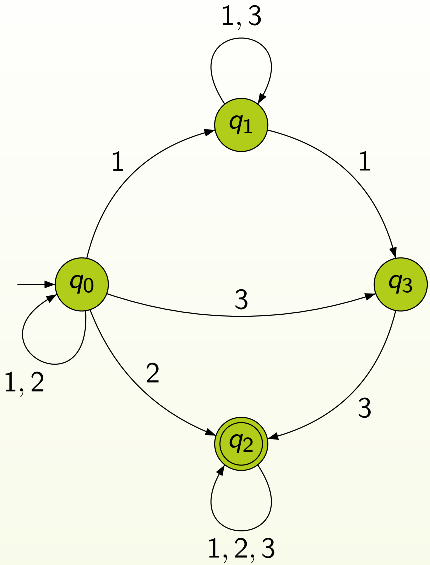
$$\{uba, ubb \mid u \in \{a, b\}^*\}.$$

NFA: example 2

$\Sigma = \{1, 2, 3\}$

δ States	Σ 1	2	3
q_0	$\{q_0, q_1\}$	$\{q_0, q_2\}$	$\{q_3\}$
q_1	$\{q_1, q_3\}$	$\emptyset$	$\{q_1\}$
q_2	$\{q_2\}$	$\{q_2\}$	$\{q_2\}$
q_3	$\emptyset$	$\emptyset$	$\{q_2\}$

$S = \{q_0\}$
$F = \{q_2\}$



Every NFA can be simulated by a DFA.

In fact, there is an algorithm which converts an NFA N into an equivalent DFA M, that is $L(M) = L(N)$.

Idea: Create potentially a state in M for every subset of states of N. In the worst case, if N has n states, then M has 2^n states.

Comment: Many of these states are not reachable so the algorithm often terminates with a smaller DFA than the worst case.

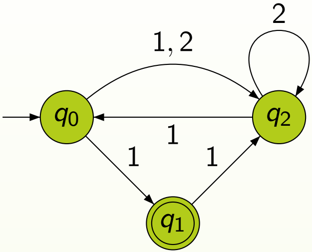
Algorithm: `NFAtoDFA` is a method for constructing a DFA equivalent with a given NFA.

Theorem: A language is regular if and only if it is recognised by an NFA.

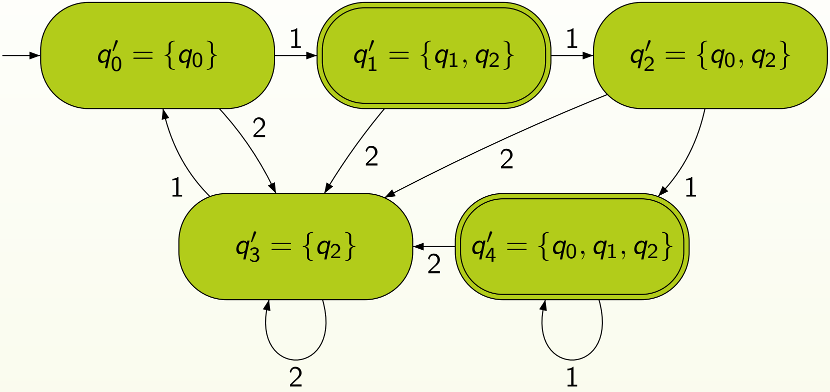
Input: NFA $N = (Q, \Sigma, \delta, S, F)$
Output: DFA $M = (Q_M, \Sigma, \delta_M, s_M, F_M)$

- The set of states of M is the set of all subsets of Q, $Q_M = 2^Q$.
- The transition from a set of states A on an element $x \in \Sigma$ is the set of all states produces by N on each pair (q, x) with $q \in A$, $\delta_M(A, x) = \{\delta(q, x) \mid q \in A\}$.
- The initial state s_M of M is the set of all initial states of N, $s_M = S$.
- The accepting states F_M of M is the set of states that have an accepting state of N, $F_M = \{A \subseteq Q \mid A \cap F \neq \emptyset\}$.

NFAtoDFA: an example 1



The NFA N



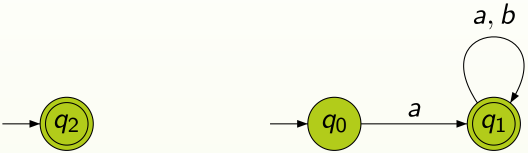
Equivalent DFA M

- The union of two regular languages is also regular.
  Proof: Given two NFAs N_A, N_B with no common states such that $A = L(N_A), B = L(N_B)$, the NFA N consisting of the union of all components of N_A, N_B recognises $A \cup B$.

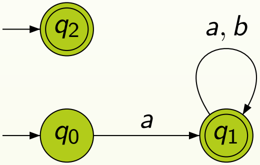
  More precisely, if $N_A = (Q_A, \Sigma, \delta_A, S_A, F_A)$ and $N_B = (Q_B, \Sigma, \delta_B, S_B, F_B)$ with $Q_A \cap Q_B = \emptyset$, then $A \cup B$ is recognised by the NFA

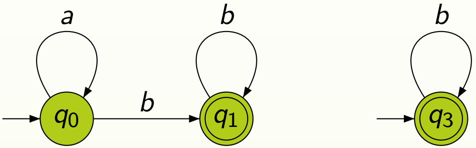
$$N = (Q_A \cup Q_B, \Sigma, \delta_A \cup \delta_B, S_A \cup S_B, F_A \cup F_B).$$

- The intersection of two regular languages is also regular.
  Proof: $A \cap B = \overline{\overline{A} \cup \overline{B}}$.


q_2
q_0
a
q_1
a, b

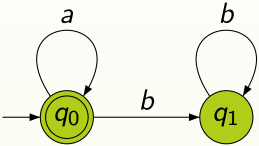
Closure under union: an example 2



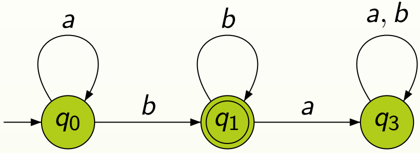


NFA N_1

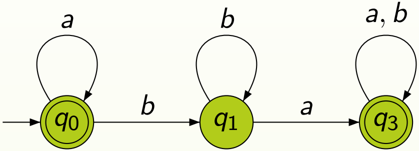
NFA N_2



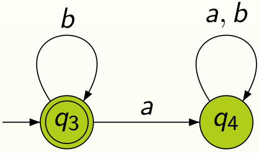
NFA accepting the complement of N_1?



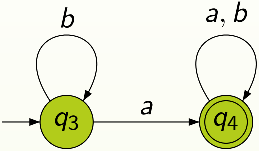
DFA M_1 equivalent to N_1



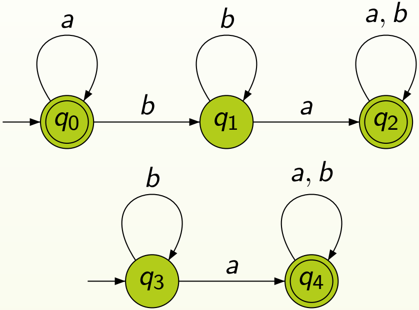
DFA $\overline{M}_1$ recognising the complement of M_1



DFA M_2 equivalent to N_2



DFA $\overline{M}_2$ recognising the complement of M_2



NFA N_3 recognising $L(\overline{M}_2) \cup L(\overline{M}_2)$

Last two steps:

- Construct a DFA M_3 equivalent to the NFA N_3
- Construct the complement of $L(M_3) = L(N_1) \cap L(N_2) = \{b^k \mid k \geq 1\}$

Recap:

- $L(N_1) = \{a^n b^m \mid n \geq 0, m \geq 1\}$
- $L(N_2) = \{b^m \mid m \geq 0\}$
- $L(M_3) = \{b^k \mid k \geq 1\}$

The closure (or Kleene star) of a language A, denoted by A^*, is the set of all strings that can be formed by concatenating together any finite number of strings of A.

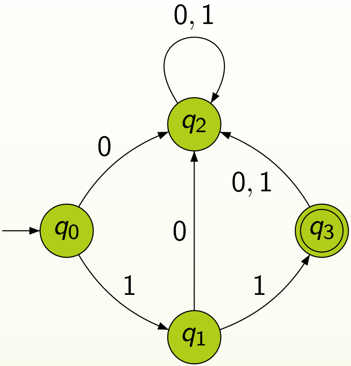
Examples:

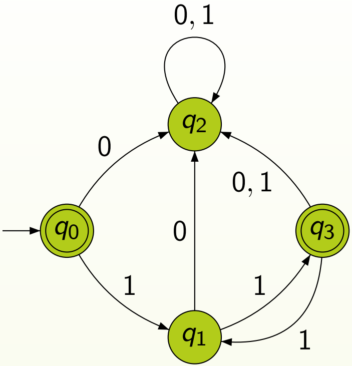
- $\{a\}^* = \{\varepsilon, a, aa, aaa, \ldots, a^n, \ldots\}$
- $\{a, ab\}^* = \{\varepsilon, a, ab, aa, abab, aab, aba, \ldots\}$

- The Kleene star of a regular language is also regular.
  Proof: Given an NFA N_A that recognizes a language A we can build an NFA N_{A^*} that recognises the closure of A by making a start state accept state and, adding transitions, with corresponding labels, from all accept state(s) to the neighbours of the initial state(s).

  Is the proof correct?

Closure operation: an example





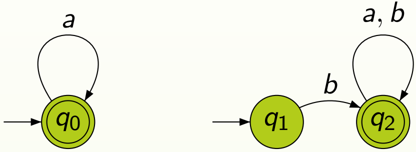
The concatenation of two languages A, B is defined to be the set of strings that can be formed by concatenating all strings of A with all strings of B, i.e.

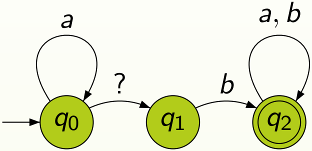
$$AB = \{xy \mid x \in A, y \in B\}.$$

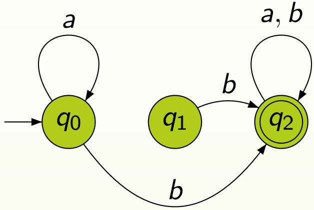
Example: If $A = \{a^n \mid n \geq 0\}$ and $B = \{bw \mid w \in \{a, b\}^*\}$, then

$$AB = \{a^n bw \mid w \in \{a, b\}^*, n \geq 0\} = \{ubv \mid u, v \in \{a, b\}^*\}.$$

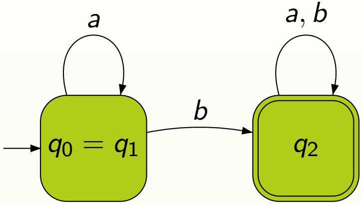
Closure under concatenation: an example 1.1



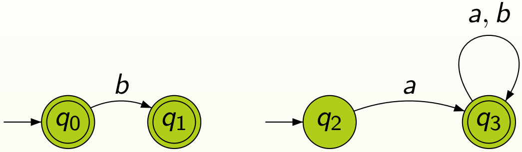

a
a, b
?
b
q_0
q_1
q_2


a
a, b
b
q_0
q_1
q_2
b

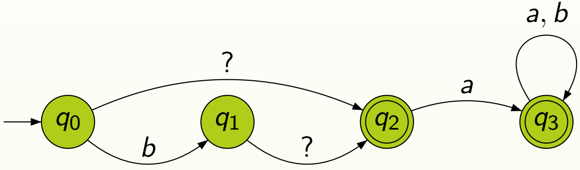
Closure under concatenation: an example 1.3'



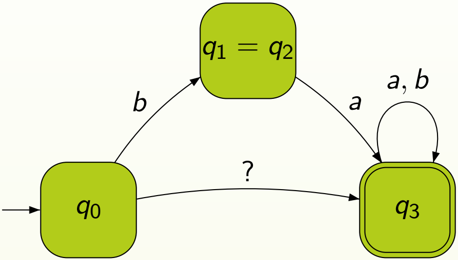
Closure under concatenation: an example 2.1

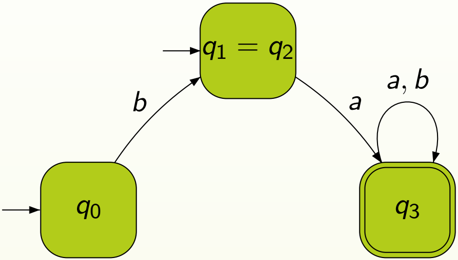


Closure under concatenation: an example 2.2



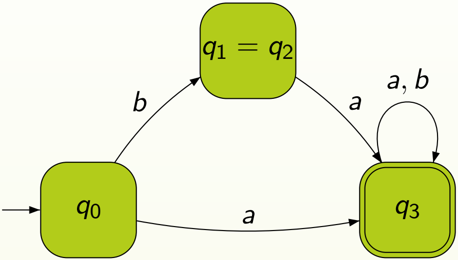
Closure under concatenation: an example 2.3





$q_1 = q_2$
b
a
a, b
q_0
q_3

Closure under concatenation: an example 2.4'



- The concatenation of two regular languages is also regular.
  Proof: Given two NFAs $N_A = (Q_A, \Sigma, \delta_A, S_A, F_A)$ and $N_B = (Q_B, \Sigma, \delta_B, S_B, F_B)$, $Q_A \cap Q_B = \emptyset$, recognising the languages A, B, respectively, we can build an NFA $N = (Q, \Sigma, \delta, S, F)$ that recognises the concatenation of A and B as follows:
  - $Q = Q_A \cup Q_B$
  - $S = S_A \cup S_B$ if one state of S_A is a final state; otherwise, $S = S_A$
  - $F = F_B$
  - $$\delta(q, c) = \begin{cases} \delta_A(q, c), & \text{if } q \in Q_A \setminus F_A, \\ \delta_B(q, c), & \text{if } q \in Q_B \setminus S_B, \\ \delta_A(q, c) \cup \{\delta_B(q', c) \mid q' \in S_B\}, & \text{if } q \in F_A. \end{cases}$$

Let A be a language and $n \geq 1$. We define:

$$A^n = \{x_1 x_2 \cdots x_n \mid x_1, x_2, \ldots, x_n \in A\}.$$

- If A is a regular language, then for each $n \geq 1$, A^n is also regular.
  Proof: $A^1 = A, A^2 = AA, \ldots, \underbrace{A^n = AA \cdots A}_{n \text{ times}}$, so the result follows from the closure under concatenation.

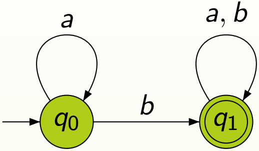
- It is algorithmically decidable whether two DFAs accept the same language.

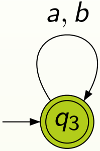
  Proof: If A, B are two languages recognised by the DFAs M_A, M_B, respectively, then (using the closure properties of regular languages) we can construct a DFA M such that:

  $$L(M) = A \,\Delta\, B = (A \cap \overline{B}) \cup (B \cap \overline{A}),$$

  and then use the equivalence:

  $$A = B \Leftrightarrow A \,\Delta\, B = \emptyset.$$



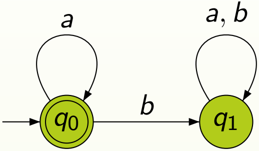


DFA M_1

$\{a^n bu \mid n \geq 0, u \in \{a, b\}^*\}$

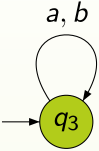
DFA M_2

$\{a, b\}^*$



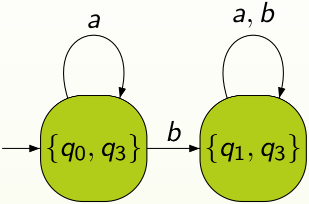
DFA $\overline{M_1}$

$\{a^n \mid n \geq 0\}$



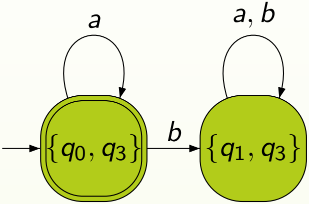
DFA $\overline{M_2}$

$\emptyset$



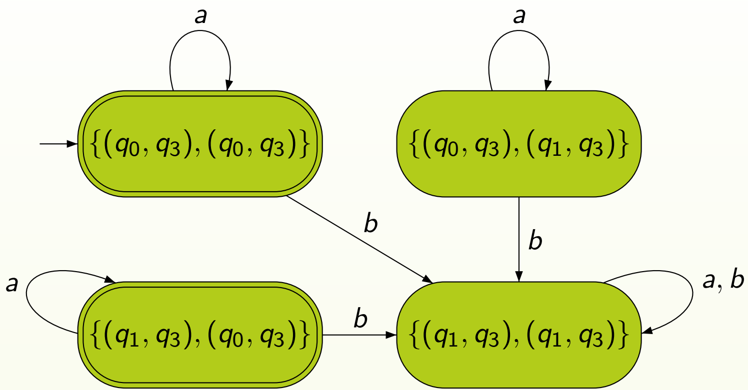
DFA $M_1 \cap \overline{M_2}$

$\emptyset$



DFA $\overline{M_1} \cap M_2$

$\{a^n \mid n \geq 0\}$



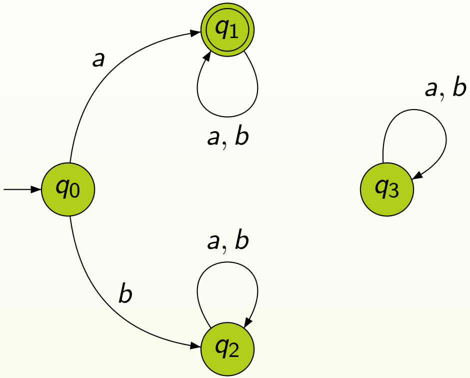
DFA $M_1 \Delta M_2$: $\{a^n \mid n \geq 0\} \neq \emptyset$ implies $L(M_1) \neq L(M_2)$

- It is algorithmically decidable whether a DFA M accepts only one a string w.
  Proof: Take $A = L(M)$ and $B = \{w\}$.
- It is algorithmically decidable whether the language accepted by a DFA M includes the language accepted by a DFA M'.
  Proof: We use the equivalence

$$L(M) \subseteq L(M') \Leftrightarrow L(M) \cap L(M') = L(M).$$

We want to minimise the number of states of a DFA, i.e. given a DFA M produce a new DFA M' such that:

- $L(M) = L(M')$,
- M' has less states than M.



The state q_3 can be removed without modifying the accepted language

From a DFA

$$M = (Q, \Sigma, \delta, s, F)$$

and any state $q \in Q$ we define the new DFA

$$M_q = (Q, \Sigma, \delta, q, F)$$

by simply replacing the initial state s with q.

We say two states p and q of M are distinguishable (k-distinguishable) if there exists a string $w \in \Sigma^*$ (of length k) such that exactly one of M_p or M_q accepts w.

If there is no such string w then we say p and q are equivalent.

Questions:

- Does there exist an algorithm deciding whether two states p and q are distinguishable?
- Does there exist an algorithm deciding whether two states p and q are k-distinguishable?
- Does there exist an algorithm deciding whether two states p and q are equivalent?

If a DFA M has two equivalent states p and q, then one of these states can be eliminated without modifying the accepted language, hence we can construct a smaller DFA M′ such that $L(M) = L(M')$.

Proof: Assume $M = (Q, \Sigma, \delta, s, F)$ and $p \neq s$. We create an equivalent DFA

$$M' = (Q \setminus \{p\}, \Sigma, \delta', s, F \setminus \{p\}),$$

where δ' is δ with all instances of $\delta(q_i, c) = p$ replaced with $\delta'(q_i, c) = q$, and all instances of $\delta(p, c) = q_i$ deleted.

The resulting automaton M' is deterministic and accepts $L(M)$.

Two states p and q are k-distinguishable if and only if for some $c \in \Sigma$, the states $\delta(p, c)$ and $\delta(q, c)$ are $(k-1)$-distinguishable.

Proof: Consider all strings $w = cw'$ of length k. If $\delta(p, c)$ and $\delta(q, c)$ are $(k-1)$-distinguishable by some string w', then p and q must be k-distinguishable by w.

Likewise, if p and q are k-distinguishable by w, then there exist two states $\delta(p, c)$ and $\delta(q, c)$ that are $(k-1)$-distinguishable by the shorter string w'.

The algorithm `minimizeDFA` finds the equivalent states of a DFA $M = (Q, \Sigma, \delta, s, F)$. It defines a series of equivalence relations $\equiv_0$, $\equiv_1$, ... on the states of Q:

$p \equiv_0 q$	if both p and q are in F or both not in F.
$p \equiv_{k+1} q$	if $p \equiv_k q$ and, for each $c \in \Sigma$, $\delta(p, c) \equiv_k \delta(q, c)$.

It stops generating these equivalence classes when $\equiv_n$ and $\equiv_{n+1}$ are identical.

Is the algorithm correct?

Distinguish lemma guarantees no more non-equivalent states.

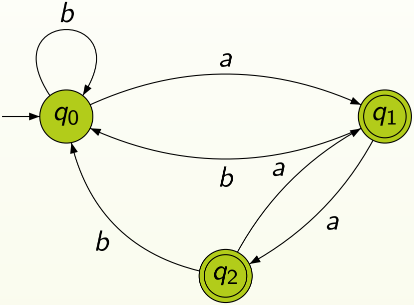
Since there can be at most the number of states non-equivalent states, the number of equivalence relations $\equiv_k$ generated cannot be larger than the number of states.

We can eliminate one state from M (using the elimination lemma) whenever there exist two states p and q such that $p \equiv_n q$.

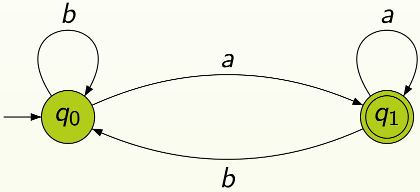
Is `the algorithm minimizeDFA` optimal?

???

The DFA M is not minimal as:
$\equiv_0 = \{\{q_0\}, \{q_1, q_2\}\}$,
$q_1 \equiv_1 q_2$,
$\equiv_1 = \{\{q_0\}, \{q_1, q_2\}\}$,
$\equiv_0 = \equiv_1$
because
$\delta(q_1, a) = q_2 \equiv_0 \delta(q_2, a) = q_1$,
$\delta(q_1, b) = q_0 \equiv_0 \delta(q_2, b) = q_0$

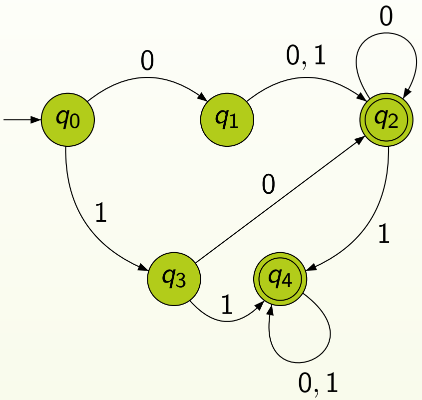


The following DFA is minimal and equivalent to M:

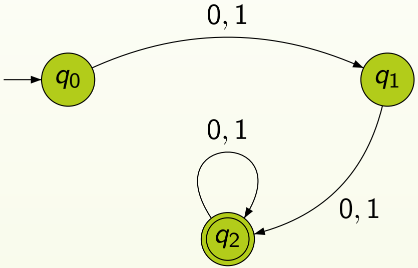


The DFA M is not minimal as:
$\equiv_0 = \{\{q_0, q_1, q_3\}, \{q_2, q_4\}\}$,

$\equiv_1 = \{\{q_0\}, \{q_1, q_3\}, \{q_2, q_4\}\}$,
$\equiv_2 = \equiv_1$,
because
$\delta(q_2, 0) = q_2 \equiv_0 \delta(q_4, 0) = q_4$,
$\delta(q_2, 1) = q_4 \equiv_0 \delta(q_4, 1) = q_4$,
$\delta(q_0, 0) = q_1 \not\equiv_0 \delta(q_1, 0) = q_2$,
$\delta(q_0, 0) = q_1 \not\equiv_0 \delta(q_3, 0) = q_2$,
$\delta(q_1, 0) = q_2 \equiv_0 \delta(q_3, 0) = q_2$,
$\delta(q_1, 1) = q_2 \equiv_0 \delta(q_3, 1) = q_4$



The following DFA is minimal and equivalent to M:



Consider the languages:

$A = \{0^m 1^n \mid n, m \geq 0\}$,
$B = \{0^m 1^m \mid m \geq 0\}$,
$C = \{w \in \{0,1\}^* \mid w$ has an equal number of 0s and 1s$\}$,
$D = \{w \in \{0,1\}^* \mid w$ has an equal number of occurrences of 01 and 10 as substrings$\}$.

Which languages are regular? Why?

Theorem 1.70. If A is a regular language, then there is a number p (the pumping length) such that every string $s \in A$ of length at least p can be written in the form

$$s = xyz$$

such that the following three conditions are satisfied:

1. for each $i \geq 0$, $xy^iz \in A$,
2. $|y| > 0$,
3. $|xy| \leq p$.

Let $M = (Q, \Sigma, \delta, q_1, F)$ be a DFA recognising A and p be the number of states of M.
Let $s = s_1 s_2 \dots s_n \in A$ with $n \geq p$ and consider the sequence of states

$$r_1 = \delta(q_1, s_1), r_2 = \delta(r_1, s_2), \dots, , r_{i+1} = \delta(r_i, s_{i+1}), \dots,$$
$$r_{n+1} = \delta(r_n, s_{n+1}).$$

As M has p states, $p + 1 \leq n + 1$, so there exist $1 \leq j < l \leq p + 1$ such that $r_j = r_l$. Split s as follows:

$$s = s_1 s_2 \dots s_n = (s_1 \dots s_{j-1})(s_j \dots s_{l-1})(s_l \dots s_n),$$

and put

$$x = s_1 \dots s_{j-1}, y = s_j \dots s_{l-1}, z = s_l \dots s_n.$$

Because s is in A, r_{n+1} is in F. For every $i \geq 0$, the trace of $xy^i z$ takes x from r_1 to r_j, continues with y taking r_j to $r_l = r_j$ i times, and finally taking z from r_j to $r_{n+1} \in F$, so M accepts $xy^i z$. (What happens when $i = 0$?)

From $j < l$ we deduce that $|y| > 0$.

As $l \leq p + 1$, $|xy| \leq p$.

Example 1.74. The language $C = \{w \in \{0,1\}^* \mid w \text{ has an equal number of 0s and 1s}\}$ is not regular.

Assume by contradiction that C is regular and let p be the pumping length. Choose the string $s = 0^p1^p$ in C; as $|s| = 2p$, it can be split as $s = xyz$ and the three conditions in the Pumping lemma are satisfied.

From the third condition we have $|xy| \leq p$, so y contains only 0s, so $xyyz$ cannot be in C.

Example 1.77. The language $E = \{0^i 1^j \mid i > j\}$ is not regular.

Assume by contradiction that E is regular and let p be the pumping length. Choose the string $s = 0^{p+1}1^p$ in E; as $|s| = 2p + 1$, it can be split as $s = xyz$ and the three conditions in the Pumping lemma are satisfied.

From the third condition we have $|xy| \leq p$, so y contains only 0s. So, xy^iz are all in E for $i \geq 0$. For $i = 0$ we get: $xz \in E$, removing at least one 0 from the original string $s = 0^{p+1}1^p$, a contradiction.

Example 173. The language $B = \{0^m 1^m \mid m \geq 0\}$ is not regular.

Assume by contradiction that B is regular and let p be the pumping length. Choose the string $s = 0^p 1^p$ in B; as $|s| = 2p$, it can be split as $s = xyz$ and the three conditions in the Pumping lemma are satisfied.
We consider three cases:

1. y contains only 0s: the string $xyyz \notin B$ because the number of 0s is not equal with the number of 1s.
2. y contains only 1s: the string $xyyz \notin B$ because the number of 0s is not equal with the number of 1s.
3. y contains both 0s and 1s: the string $xyyz \notin B$ because some 0 follows a 1.

Problem 1.48. The language $D = \{w \in \{0,1\}^* \mid w$ has an equal number of occurrences of 01 and 10 as substrings$\}$ is regular.

Observe that any binary string beginning and ending with the same digit has an equal number of occurrences of the substrings 01 and 10. Thus, $D = \{\varepsilon\} \cup \{0,1\} \cup 0\{0,1\}^*0 \cup 1\{0,1\}^*1$.

A grep pattern, also known as a regular expression, describes the text that we are looking for.

For instance, a pattern can describe words that begin with C and end in l. A pattern like this would match "Call", "Cornwall", and as well as many other words, but not "Computer".

Most characters that we type into the Find & Replace dialogue (in your favourite editor) match themselves. For instance, if you are looking for the letter "s", Grep stops and reports a match when it encounters an "s" in the text.

A range of characters can be enclosed in square brackets. For example `[a-z]` would denote the set of lower case letters. A period . is a wild card symbol used to denote any character except a newline.

The Kleene regular expressions over the alphabet Σ and the sets they designate are:

1. Any $c \in \Sigma$ is a regular expression denoting the set $\{c\}$.
2. If E_1, E_2 are regular expressions and E_1 denotes the set S_1, E_2 denotes the set S_2, then so are:
   - $E_1 + E_2$ (or $E_1|E_2$) which denotes the union $S_1 \cup S_2$,
   - $E_1 E_2$ which denotes the concatenation $S_1 S_2$,
   - E_1^* which denotes the Kleene closure S_1^*.

Sample regular expressions over $\Sigma = \{a, b, c\}$ and their corresponding sets (languages):

regular expression	denoted set (language)
a	$\{a\}$
ab	$\{ab\}$
$a + bb$	$\{a, bb\}$
$(a + b)c$	$\{ac, bc\}$
c^*	$\{\varepsilon, c, cc, ccc, \ldots\}$
$(a + b + c)cba$	$\{acba, bcba, ccba\}$
$a^* + b^* + c^*$	$\{\varepsilon, a, b, c, aa, bb, cc, aaa, bbb, ccc, \ldots\}$
$(a + b^*)c(c^*)$	$\{ac, acc, accc, \ldots, c, cc, ccc, \ldots,$ $bc, bcc, bbccc, \ldots\}$

A regular set over an alphabet Σ is either the empty set, the set $\{\varepsilon\}$, or the set of strings denoted by some regular expression.

Kleene's Theorem: Regular sets coincide with regular languages.

Proof: We will show only one implication: For any regular set L there is an NFA N such that $L(N) = L$.

- NFAs for $L = \emptyset$ and $L = \{\varepsilon\}$ are easy to construct: an NFA with no final states works in the first case and an NFA with one initial and final state and no transitions works in the second case.
- Now suppose E is a regular expression for L. We construct an NFA N such that $L(N) = L$ based on the length of E. We proceed by induction.

- Verification: If $E = \{c\}$ for some $c \in \Sigma$, then we can take $N = (Q, \Sigma, \delta, S, F)$ where $Q = \{q_0, q_1\}, S = \{q_0\}, F = \{q_1\}$ and there is one transition $\delta(q_0, c) = q_1$.
- Induction:
  - If N_1, N_2 are NFAs accepting the languages denoted by E_1 and E_2, respectively, then in view of the closure under union the NFA N_{union} accepts the language denoted by $E_1 + E_2$:

$$L(N_{union}) = L(N_1) \cup L(N_2).$$

- Induction (continued):
  - If N_1, N_2 are NFAs accepting the languages denoted by E_1 and E_2, respectively, then in view of the closure under concatenation the NFA $N_{concatenation}$ accepts the language denoted by E_1E_2:

$$L(N_{concatenation}) = L(N_1)L(N_2).$$

  - If N_1 is a NFA accepting the language denoted by E_1, then in view of the closure under Kleene closure the NFA N_* accepts the language denoted by E_1^*:

$$L(N_*) = L(N_1)^*.$$

Construct an NFA accepting exactly the language denoted by the regular expression: $(01)^* + 1$.
We use the closure properties of regular languages:

- construct NFAs N_1 and N_2 accepting the languages $\{0\}$ and $\{1\}$, respectively
- construct an NFA N_3 for the concatenation of $L(N_1)$ and $L(N_2)$ obtaining the language $\{01\}$
- construct an NFA N_4 for the Kleene closure of $L(N_3)$ so obtaining $\{01\}^*$
- construct an NFA N_5 for the union of $L(N_4)$ and $L(N_2)$ obtaining the language $\{01\}^* \cup \{1\}$
- we may want to transform N_5 into an equivalent *DFA* (also minimise it)

- Construct a regular expression denoting the language:

$$A = \{0^n 1^m \mid n, m \geq 0\}.$$

The language L is regular and

$$\begin{aligned} A &= \{0^n 1^m \mid n, m \geq 0\} \\ &= \{0^n \mid n \geq 0\}\{1^m \mid m \geq 0\} \end{aligned}$$

so A is denoted by 0^*1^*.

- There is no a regular expression denoting the language:

$$B = \{0^n 1^n \mid n \geq 0\}$$

because B is not regular.

There is no a regular expression denoting the language:

$$C = \{uuww \mid u, w \in \{a, b\}^*\}$$

because C is not regular. Prove this fact!

The *pattern matching problem*:

> *Given a (short) pattern P and a (long) text T, (over an alphabet Σ) determine whether P appears somewhere in T.*

Example: If $P = aba$ and $T = baababababaaaba$, then the first occurrence of P in T appears at the third character:

$$T = ba\mathbf{aba}babaaaba$$

Of course, there are some other occurrences.

Try each possible position the pattern $P[1..m]$ could appear in the text $T[1..n]$:

```
for (i=0; T[i] != '\0'; i++)
{
        for (j=0; T[i+j] != '\0' && P[j] != '\0'
                        && T[i+j]==P[j]; j++) ;
        if (P[j] == '\0') found a match
}
```

There are two nested loops; the inner one takes $O(m)$ iterations and the outer one takes $O(n)$ iterations so the total time is the product, $O(mn)$. This is slow!

An example: if $T[1..n]$ is a^n, and $P[1..m]$ is b^m, then it takes m comparisons each time to discover that we don't have a match, so mn overall.

The worst case scenario may not be too frequent because the inner loop usually finds a mismatch quickly and moves on to the next position without going through all m steps.

Can we do it better?

Solution: Consider the language

$$A(P) = \{x \mid x \text{ contains the pattern } P\}.$$

Assume that $A(P)$ is regular! Let M be a DFA for $A(P)$. When processing an input M must enter an accepting state when it has just finished 'seeing' the first occurrence of P, and thereafter it must remain in some accepting state or other.

Is $A(P)$ regular?

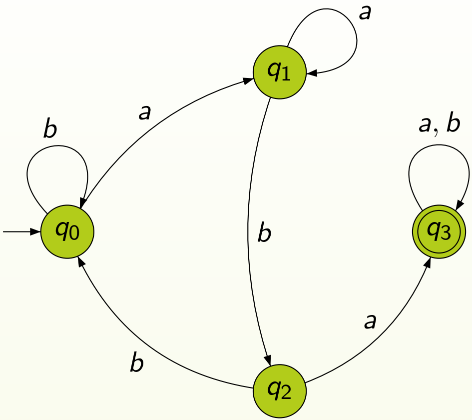
Answer: yes.

Example: If $P = aba$ and the alphabet is $\{a, b\}$, then

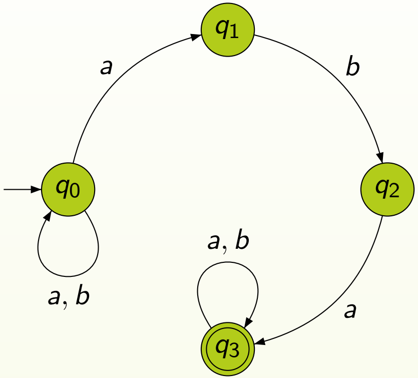
$$A(P) = \{x \in \{a, b\}^* \mid x = uPv, \text{ for some } u, v \in \{a, b\}^*\},$$

or

$$A(P) = \{uabav \mid u, v \in \{a, b\}^*\}.$$



A DFA for $AP(aba)$



An NFA for $A(aba)$

For every string P, the language

$$A(P) = \{uPv \mid u, v \in \{a, b\}^*\}$$

is regular.

Proof: Let M be a DFA recognising exactly $\{P\}$. An NFA recognising $A(P)$ can be obtained from a DFA M by adding loops labelled with a and b to the initial and final states of M.

Is the fact that $A(P)$ is regular of any use?

Yes, because there is an algorithm testing the membership problem for $A(P)$ which is the same as testing whether P appears in the input text T.

How complex is this algorithm?

In the practice of computing regular expressions (abbreviated as regex or regexp, with plural forms regexes) differ from the Kleene definition discussed before.

Regexes are written in a formal language that can be interpreted by a regular expression processor, a program that either serves as a parser generator or examines text and identifies parts that match the provided specification.

There are various versions of regexes; they provide an expressive power that exceeds the regular languages.

Here is an example. Regexes have the ability to group sub-expressions with parentheses and recall the value they match in the same expression.

Using this feature one can write a pattern that matches strings of repeated words like "papatoetoe" (squares). The regex to match "papatoetoe" is

$$(.^*)\backslash 1(.^*)\backslash 2,$$

where $\backslash 1 =$pa and $\backslash 2 =$toe were the sub-matches. The language associated to this pattern is not regular.

More examples and testers at http://regexlib.com.